Michael Klein
Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5
76128 Karlsruhe
kleinm@ipd.uni-karlsruhe.de

Handbuch zur

DIANE Service Description

Version: 1.0

Technischer Bericht der Fakultät für Informatik
Universität Karlsruhe (TH)
TR 2004-17, ISSN 1432-7864

Neueste Version unter: http://www.ipd.uka.de/DIANE/docs/DSD-Cookbook.pdf

Versionen dieses Dokuments

- Version 1.0: Version, die als Technischer Bericht veröffentlicht wurde. Verschiedene Fehlerkorrekturen. Schluss und Anhang hinzugefügt.
- Version 0.8: Kapitel 11 „Definieren von Groundings“ hinzugefügt.
- Version 0.7: Kapitel 10 „Vorbedingungen“ hinzugefügt.
- Version 0.6: Kapitel 9 „Unterschiede zwischen Dienstangeboten und -anfragen“ hinzugefügt.
- Version 0.5.2: Umstrukturierung in 5 Teile. Umstrukturierung des Aufbaus von Teil III.
- Version 0.5.1: Umbenennung von `domain.general` in `domain.measure`. (Abschnitt 7.3 und Anhang A.8).
- Version 0.5: Kapitel 8 „Allgemeiner Prozess zum Aufbau von Dienstbeschreibungen“ hinzugefügt. Anhang hinzugefügt.
- Version 0.4.2: Änderungen an der Typvergleichsstrategie (Abschnitte 4.1.6 und 4.2.4).
- Version 0.4.1: Korrekturen an primitiven Werten für `Date` und `DateTime` (Abschnitt 3.1).
- Version 0.4: Kapitel 7 „Allgemeines zur Beschreibung von Diensten“ hinzugefügt.
- Version 0.3.2: Semantik von `compareTo` und `fCompareTo` an Java-Standard (Interface `Comparable`) angepasst (Abschnitte 2.1.3 und 2.1.4).
- Version 0.3.1: valueclass/entityclass/public für j-dsd angepasst (Abschnitte 3.3 und 3.4).
- Version 0.3: Kapitel 6 „Variablen“ hinzugefügt.
- Version 0.2.1: ConnectingStrategy in j-dsd geändert (Abschnitt 4.1.5).
- Version 0.2: Kapitel 5 „Mengen“ hinzugefügt.
- Version 0.1: Initiale Version mit Kapiteln 1-4

Inhaltsverzeichnis

Teil I
Übersicht über DSD

1 Einführung in DSD

Die DIANE Service Description (DSD) ist eine Sprache, die entwickelt wurde, um elektronische Dienste semantisch eindeutig beschreiben zu können. Motivation für die Entwicklung von DSD war die vollständige Automatisierung der Dienstnutzung, d.h. beginnend mit einem Funktionsaufruf soll es DSD ermöglichen, automatisch einen geeigneten Dienstgeber zu finden und diesen korrekt aufzurufen. Im Detail sieht der Ablauf wie folgt aus: Ein Anwender oder eine Anwendung stellt den *Dienstnehmer* dar. Er registriert sich für Funktionen, welche im Folgenden wiederholt und in unterschiedlicher Parametrisierung genutzt werden sollen. Hierzu beschreibt er die benötigte Funktionalität semantisch mittels DSD. Auch Anbieter von Funktionalität (*Dienstgeber*) beschreiben mittels DSD, welche Dienste sie öffentlich zur Verfügung stellen. Ruft nun der Dienstnehmer eine seiner beschriebenen Funktionen konkret auf, so wird die zugehörige Anfragebeschreibung automatisch (d.h. ohne menschliche Hilfe) mit allen im System verfügbaren Angebotsbeschreibungen verglichen. Hierbei passt ein Angebot zu einer Anfrage, wenn dieses die gewünschte Funktionalität erbringen kann und sich im Rahmen der in der Anfrage spezifizierten Abweichungen bewegt. Dabei ist es nicht zwingend nötig, dass der angeforderte und der angebotene Dienst eine syntaktische gleiche Beschreibung oder identische Schnittstellen besitzen. Aus den passenden Angeboten wird ohne weitere Rückfragen ein geeigneter Dienstgeber ausgewählt. Dieser wird anschließend automatisch und korrekt parametrisiert aufgerufen.

1.1 Grundkonzepte von DSD

Es wird klar, dass eine geeignete Dienstbeschreibung eine wesentliche Rolle bei der Umsetzung eines solchen Ablaufs spielt. Sie muss so viel Information enthalten, dass der Vergleich, die Auswahl und der Aufruf von Diensten ohne menschliche Hilfe durchgeführt werden kann. Hierzu wurden in DSD folgende grundlegenden Konzepte verwendet:

- Vereinheitlichung des Vokabular durch Verwendung von Ontologien
- Reine Zustandsorientierung
- Strukturierung der Anfragen durch Ontologieschichtung
- Vollständige Präferenzintegration in Anfragen

Diese sollen im Folgenden vorgestellt werden.

1.1.1 Vereinheitlichung des Vokabulars

Ein wichtiger Ansatz in DSD wie auch in anderen Beschreibungssprachen ist die Vereinheitlichung des verwendeten Vokabulars. Dies ist nötig, um eine grundlegende Verständlichkeit der Beschreibungen durch einen Rechner zu erlangen, welche für eine sinnvolle, automatische Verarbeitung (insbesondere für den Vergleich) zwingend nötig ist. Bei der Vereinheitlichung von Vokabular hat sich der Einsatz von Ontologien bewährt. Ontologien sind Beschreibungen der Welt, auf die sich die Gruppe

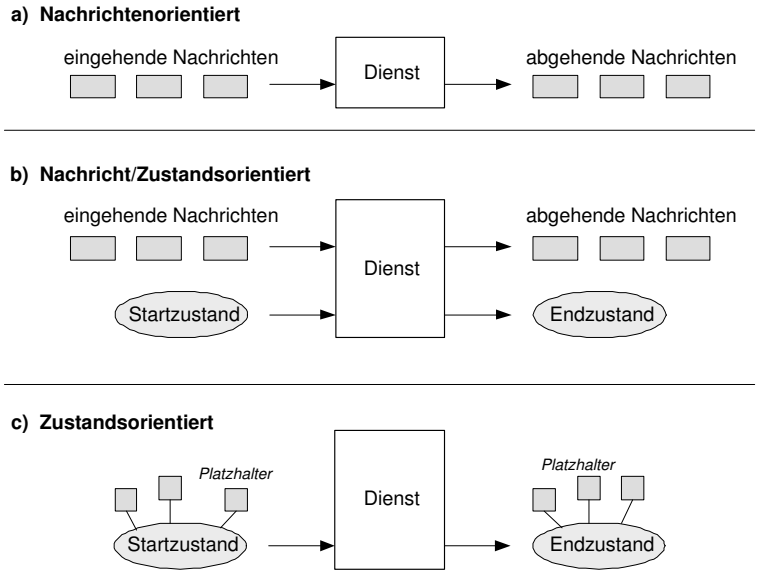


Abbildung 1: Verschiedene Arten von Dienstbeschreibungssprachen: (a) Nachrichtenorientierte Beschreibungssprachen versuchen Dienste durch ihre eingehenden und abgehenden Daten zu beschreiben, (b) Gemischte Nachrichten/Zustands-Beschreibungssprachen drücken zusätzlich die funktionale Semantik durch die Angabe eines Start- und eines Endzustands aus, wogegen (c) rein zustandsorientierte Beschreibungssprachen zugunsten konfigurierbarer Platzhalter in der Zustandsbeschreibung auf die Angabe von Nachrichten verzichten.

ihrer Verwender (die *Community*) geeinigt hat (vgl. [3]). Durch diese Einigung wird erreicht, dass Personen, die über denselben Sachverhalt sprechen, auch dieselben Worte hierfür verwenden, was es einem Rechner ermöglicht, die Aussagen z.B. semantisch korrekt vergleichen zu können.[1]

Für Ontologien existieren in der Literatur eine Reihe von Ausdrucksformen. Weit verbreitet und auch in DSD verwendet ist eine Auftrennung der Beschreibung in einen Teil zur Beschreibung des Schemas (oder terminological box/T-Box in der Sprache der Beschreibungslogik [1]), welches abstrakt die grundlegende Klassen und ihre möglichen Beziehungen auflistet, und einen Teil zur Beschreibung konkreter Individuen (die assertion box/A-Box), in welcher Instanzen dieser Klassen definiert sind, die stellvertretend für Individuen der realen Welt stehen. Zur Beschreibung von Schema und Instanzen existiert in der Regel ein Metaschema (auch Ontologiebeschreibungssprache), welches die syntaktischen Bausteine definiert, die zur Definition von Klassen, Beziehungen, Instanzen etc. benötigt werden. In DSD sind das die so genannten *DSD Elements* .

1.1.2 Reine Zustandsorientierung

Betrachtet man existierende Ansätze zur Beschreibung von Diensten, so stellt man fest, dass häufig nur der Datenfluss des Diensten, d.h. seine Ein- und Ausgaben beschrieben sind (siehe Abbildung 1a). Bekannte Beispiele hierfür sind WSDL [18] und ebXML [15]. Solche Beschreibungen sind jedoch für eine automatische Verarbeitung nicht geeignet. Einerseits bieten sie keinen nutzbaren Mehrwert im Vergleich zur Angabe der Schnittstelle, bei der die erbrachte Funktionalität aus dem Nachrichtenfluss

[1] In der Literatur existieren noch eine Reihe weiterer Definitionen für Ontologien, die teilweise auf die Einigung durch eine Gruppe verzichten. Hierbei wird die Vereinheitlichung durch einen voran gestellten Schritt des Ontologie-Matchings erreicht, in dem semi-automatisch unterschiedlich benannte Konzepte für den gleichen Sachverhalt gefunden werden.

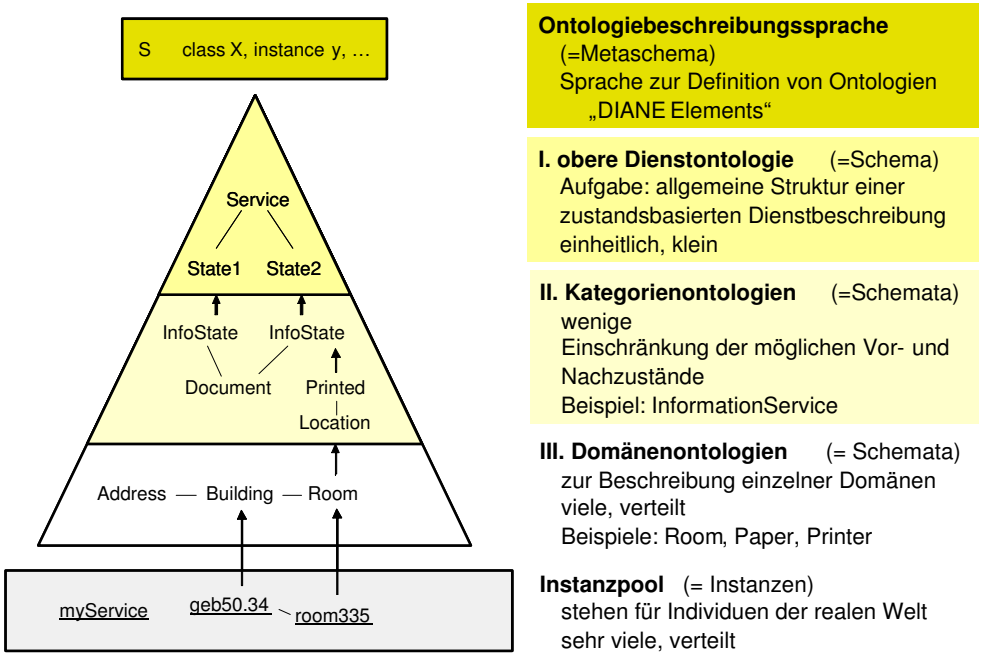


Abbildung 2: Schichtung von Ontologien in DSD.

erraten werden muss. Andererseits kann eine solche Beschreibung nicht vermitteln, wenn Dienstnehmer und Dienstgeber unterschiedliche Nachrichten verlangen bzw. liefern (abgesehen von trivialen Umstellungen bzw. Umbenennungen). Auch eine Hinzunahme einer getrennten Beschreibung der Zustände vor und nach der Dienstausführung ist nicht ausreichend (siehe Abbildung 1b). Dies wird beispielsweise in OWL-S [17] oder ICL [13] eingesetzt. Unklar bleibt hier, wie sich die ausgetauschten Nachrichten auf die involvierten Zustände auswirkt. Auch die Probleme mit unterschiedlichen Erwartungen an die Nachrichten bei Dienstnehmer und Dienstgeber bleiben bestehen.

In DSD wird daher das Konzept der *reinen Zustandsorientierung* verfolgt und ganz auf eine explizite Beschreibung von Nachrichten verzichtet (siehe Abbildung 1c). DSD beschreibt daher nur noch den benötigten Zustand vor Beginn sowie den erreichten Zustand nach der erfolgreichen Dienstausführung (siehe dazu auch [11]). Teile dieser Zustandsbeschreibungen sind dabei bereits bei der Aufstellung der Dienstbeschreibung bekannt und werden durch Instanzen beschrieben, noch unbekannte Teile werden durch Variablen ausgedrückt, die im Verlauf der Dienstnutzung von Dienstnehmer und -geber ausgefüllt werden müssen. Hierdurch ist der Einfluss der Parameter auf das Ergebnis eindeutig ersichtlich. Wie auch alle anderen Teile der Beschreibung werden in DSD die Beschreibungen der Zustände ebenfalls aus bestimmten Ontologien entnommen.

1.1.3 Schichtung von Ontologien

Gibt man für eine Dienstbeschreibung nur vor, dass diese eine Beschreibung des Vor- und des Nachzustands enthalten soll, so erhält man zwar ein sehr flexibles Rahmenwerk, allerdings ist dies aufgrund der extremes Generizität für die Praxis nicht nutzbar. Man ist damit konfrontiert, dass Dienste unterschiedlichste Auswirkungen auf den Zustand der Welt haben und sich zudem auf die unterschiedlichsten Anwendungsgebiete beziehen können. Ohne weitere Vorgaben entstehen unstrukturierte Beschreibungen, die für eine automatische Weiterverarbeitung ungeeignet sind.

Zusätzlich zu der klassischen Aufteilung in Metaschema (durch *DSD Elements*), Schema und Instanzen, wird daher in DSD zusätzlich die Schemaebene in drei Ebenen unterteilt und einzelne Schemadefinitionen bereits fest durch DSD vorgegeben (siehe dazu auch [7]). Folgende Ebenen werden unterschieden (siehe Abbildung 2):

- **Obere Dienstontologie.** Diese definiert das Schema für das allgemeine Grundgerüst von zustandsbasierten Dienstbeschreibungen. Hier ist verankert, dass ein Dienst durch die Angabe eines Vor- und eines Nachzustands zu spezifizieren ist und weitere, nichtfunktionale Informationen (wie der Dienstname, der Anbieter, Dienstqualitäten, der technische Zugangsweg etc.) angegeben werden können. Das Schema ist sehr klein und wird einheitlich von DSD für alle Teilnehmer vorgegeben.
- **Kategorieontologien.** Aufgabe dieser Ontologien ist es, den Raum aller möglichen Dienste in Kategorien mit ähnlichen Zustandsübergängen zu unterteilen. Hierzu gibt jede dieser Ontologien im Wesentlichen Einschränkungen an die zu verwendeten Zustände vor. Mögliche Kategorien könnten Informationsdienste sein, die den Zustand eines Dokuments ändern (siehe dazu auch [12]), oder Wissensdienste, die den Zustand von Wissen ändern, oder Realweltdienste, welche den Zustand realweltlicher Gegenstände ändern. Insgesamt sind nur wenige Kategorien zu erwarten, von denen die wichtigsten bereits durch DSD vorgegeben sind. In Absprache mit der Community kann jedoch jeder Benutzer selbst weitere Kategorieschemata erstellen.
- **Domänenontologien.** Diese Ontologien dienen dazu, die unterschiedlichen Anwendungsgebiete zu formalisieren, in welchen die zu beschreibenden Dienste operieren. Hier sind tausende verschiedener Ontologien zu erwarten, die nicht durch DSD vorgegeben werden, sondern von der Community selbst eingebracht werden können. Wichtig ist, dass neben der Definition von Schemata auch Instanzen als Stellvertreter für reale Individuen definiert werden sollten, um eine Domäne vollständig zu beschreiben.

Insgesamt erreicht man durch diese Dreiteilung der Schemabeschreibung einerseits strukturierte, zustandsorientierte Dienstbeschreibungen, andererseits verliert man durch die Austauschbarkeit der Ontologien nicht die Flexibilität einer gänzlich freien Dienstbeschreibungssprache.

1.1.4 Vollständige Präferenzintegration in Anfragen

Viele der existierenden Dienstbeschreibungssprachen verwenden dieselbe Technik zur Beschreibung von Anfragen wie die zur Beschreibung von Dienstangeboten. Die Vorgehensweise hierbei ist, dass der Dienstnehmer die benötigte Funktionalität als perfekten Wunschdienst beschreibt, was dann durch einen allgemeinen Vergleicher mit den Angebotsbeschreibungen auf Ähnlichkeit verglichen wird. Hierdurch kommt es jedoch zu einem beeinflussten Vergleichsergebnis, da der Vergleicher auf vordefinierte, dem Dienstnehmer unbekannte Heuristiken zurückgreifen muss, um Abweichungen zwischen Anfrage und Angebot bewerten zu können. Man stellt fest, dass der Dienstnehmer in der Regel nicht bereit ist, ein solches Vergleichsergebnis und die zugehörige automatische Auswahl des Dienstgebers ohne Rückfragen hinzunehmen, sondern er lässt sich die besten Lösungen präsentieren, überprüft sie manuell und wählt den geeignetsten Dienstgeber aus.

Man kann das Problem lösen, wenn man versteht, dass es einen prinzipiellen Unterschied zwischen Angebots- und Anfragebeschreibungen gibt (siehe dazu auch [8, 10]):

- **Angebotsbeschreibungen.** Typischerweise kennt der Dienstanbieter alle Details seines Dienstes. Daher kann er den Dienst als einzelne Instanz beschreiben. Teile, die zum Zeitpunkt der Beschreibungserstellung noch unbestimmt sind, werden durch Mengen oder Variablen ersetzt.

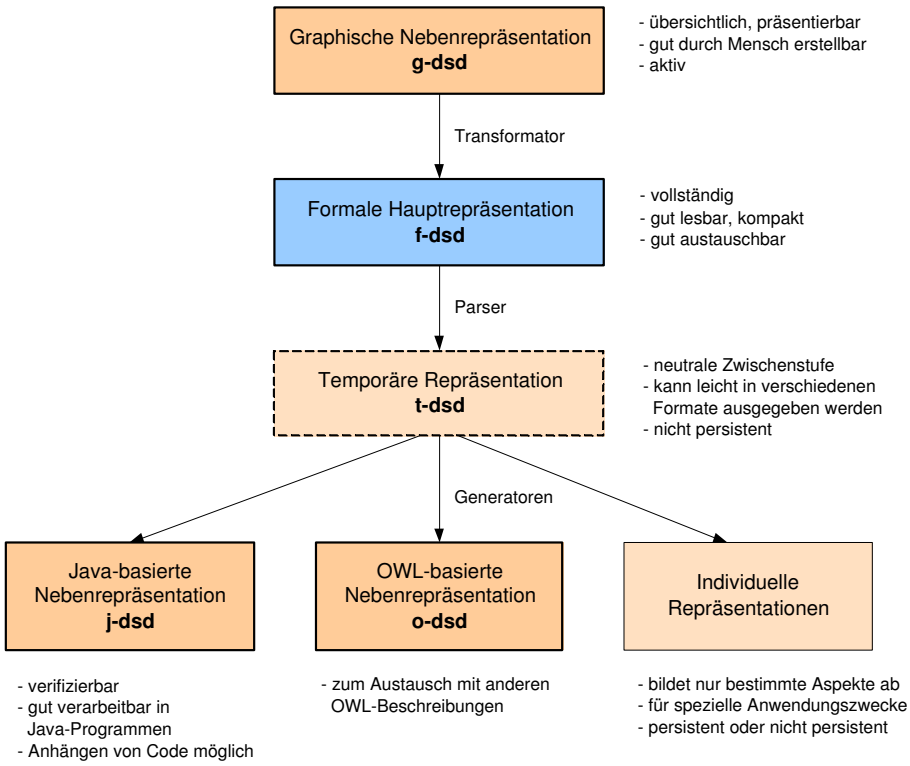


Abbildung 3: Repräsentationsformen in DSD.

- **Anfragebeschreibungen.** Der Dienstnehmer will in der Regel eine bestimmte Funktionalität erbracht wissen und denkt dabei nicht zwangsläufig an einen bestimmten, einzelnen Dienst. Typischerweise sind auch mehrere unterschiedliche Dienste zur Erbringung dieser Funktionalität geeignet. Es ist daher wenig sinnvoll, eine einzelne Dienstinstanz zu notieren. Vielmehr wäre es für ihn interessant, eine Menge geeigneter Dienste aufstellen zu können und dabei zu markieren, welche er hiervon im Zweifelsfall präferieren würde.

Die DSD trägt diesem Unterschied Rechnung, indem sie eine spezielle Anfragebeschreibung zulässt: unscharfe, deklarative Mengen. Diese erlauben es dem Dienstnehmer, seine Präferenzen bezüglich der benötigten Funktionalität sehr genau in den Anfrage zu integrieren. Wir sprechen daher von präferenzbeinhaltenden Anfragen. Der Vergleicher selbst braucht dann keine allgemeinen Heuristiken mehr zu verwenden, sondern überprüft nur, welche Mengenzugehörigkeiten die einzelnen Angebote zu der vom Benutzer aufgestellten, unscharfen Anfragemenge haben. Der Vergleicher ist dennoch eine der komplexesten Komponenten der DIANE-Architektur, was daran liegt, dass die Beschreibungen Variablen enthalten, deren Wert zum Zeitpunkt des Vergleichs nicht feststeht und die zudem vom Vergleicher mit konkreten, am besten geeigneten Werten gefüllt werden müssen. Details zum Vergleicher finden sich in [9], [14] und [16].

1.2 Repräsentationsformen von DSD

Für die DSD gibt es nicht *die* Syntax, sondern es existieren mehrere Repräsentationsformen, die jeweils unterschiedliche Vor- und Nachteile bieten und daher in verschiedenen Situationen zum Einsatz kommen. Abbildung 3 zeigt deren Zusammenhang:

- Im Mittelpunkt steht die **formale Hauptrepräsentation** (kurz f-dsd). In ihr sind die DSD Elements durch eine formale Grammatik definiert, deren Ziel es ist, möglichst kompakte und gut lesbare Beschreibungen zu erhalten. Da die Beschreibungen rein textbasiert sind, können sie leicht im System ausgetauscht werden. Als Hauptrepräsentation strebt die f-dsd nach Vollständigkeit, d.h. in ihr sind im alle Konzepte und Beschreibungsmöglichkeiten der DSD enthalten und ausdrückbar. Die Syntax der f-dsd hat Anleihen an objektorientierten Programmiersprachen und übernimmt die Darstellung primitiver Werte aus XML Schema [19]. Um eine übersichtliche Darstellung zu erhalten wurde auf Verwendung von XML verzichtet [5].

- Eine anschauliche Darstellung liefert die **graphische Nebenrepräsentation** (kurz g-dsd). Sie wurde als visuelle Sprache konzipiert, so dass sich menschliche Benutzer in ihr schnell zurecht finden, um so vorhandene Beschreibungen gut verstehen und neue Beschreibungen effizient aufbauen zu können. Weiterhin eignet sich diese Repräsentationsform dazu, Beschreibungen in Vorträgen und Papieren anschaulich präsentieren zu können. Editiert werden g-dsd-Beschreibungen mit dem Microsoft-Programm VISIO, welches durch intelligente Schablonen einem menschlichen Benutzer aktiv bei der Erstellung von Beschreibungen behilflich ist [4]. Ein in VISIO integrierter Transformator kann eine Beschreibung in g-dsd in f-dsd umwandeln. Die g-dsd ähnelt in ihrem Grundaufbau UML-Klassen- und Instanzdiagrammen, wurde jedoch um spezielle Konzepte erweitert.

- Um die formale Repräsentation verarbeiten zu können, wird sie zunächst von einem speziellen Parser eingelesen und als **temporäre Repräsentation** (kurz t-dsd) nicht-persistent im Hauptspeicher abgelegt. Diese neutrale Zwischenstufe ist im Wesentlichen darauf ausgelegt, in eine der weiteren Nebenrepräsentation persistent ausgegeben zu werden. Auch eigene Spezialrepräsentationen können hieraus leicht abgeleitet werden.

- Eine wichtige Repräsentierung ist die **Java-basierte Nebenrepräsentation** (kurz j-dsd). In ihr wird das Schema zu einem Satz von Java-Klassen, welche durch ihren Quellcode dargestellt sind. Instanzen hingegen sind gewöhnliche Java-Objekte, die durch statische Erzeugungsklassen persistent gemacht werden. Besondere Konstrukte der DSD (wie z.B. Mengen oder Variablen) werden durch spezielle Klassen und Tricks umgesetzt. Diese Darstellungsform hat eine Reihe von Vorteilen: Zunächst können durch Kompilierung des Quellcodes automatisch semantische Fehler entdeckt werden, deren Überprüfung in f-dsd komplexe Zusatzprogramme erfordert hätte. Weiterhin können die übersetzten Klassen direkt in allen Java-Programmen verwendet werden. Dies ist wichtig, da im DIANE-Projekt zur Zeit alle Komponenten, die auf Dienstbeschreibungen angewiesen sind, in dieser Programmiersprache vorliegen. Drittens ist es möglich, an bestimmte ontologische Klassen Programmlogik anzuhängen, um somit immer wiederkehrende Verarbeitungsweisen festzuhalten. Beispielsweise hat die Javaklasse zur Umsetzung einer Menge bereits eine Methode, die testet, ob eine bestimmte Instanz zu dieser Menge gehört oder nicht. Aus diesem Grund verarbeitet auch der Vergleicher Beschreibungen in dieser Darstellung.

- Zum Austausch mit anderen Forschungsgruppen eignet sich die **OWL-basierte Nebenrepräsentation** (kurz o-dsd). Diese stellt alle Konzepte der DSD mit den Mitteln der standardisierten, XML-basierten Web Ontology Language dar.[2]

- Neben den vorhandenen Darstellungsformen können auch **individuelle Repräsentationen** interessant sein. Diese sind häufig verkürzend, d.h. sie stellen nur die Aspekte einer Beschreibung dar, die für den jeweiligen Anwendungszweck von Bedeutung sind und blenden unwichtige Teile aus. Hier sind sowohl persistente als auch nicht-persistente Repräsentierungen denkbar.

[2] Anmerkung: Zur Zeit existiert noch kein Generator zur Ausgabe in o-dsd.

1.3 Aufbau des Handbuchs und verwendete Schriften

Das vorliegende Handbuch gliedert sich in drei Teile: Teil I beinhaltet diese Einleitung. In Teil II wird erläutert, wie Schemata, Instanzen, Mengen und Variablen in DSD definiert werden können. Hierzu werden die drei Repräsentationsformen g-dsd, f-dsd und j-dsd eingeführt. Teil III behandelt dann, wie die Konstrukte genutzt werden können, um Dienste zu beschreiben. Im Anhang finden sich die wichtige Ontologien sowie die Grammatik von f-dsd.

Abschnitte, die mit (*) gekennzeichnet sind, sind noch aktueller Forschungsgegenstand und können sich noch ändern.

Im Handbuch werden die folgenden Schriften verwendet:

- *Kursive Schrift* zur Einführung neuer Begriffe. Diese sind auch zum Großteil im Index zu finden.
- Serifenlose Schrift zur Beschreibung von Klassen-, Attribut- und Instanznamen sowie Variablenkategorien.
- `Schreibmaschinenschrift` zur Beschreibung von Java- oder f-dsd-Code, Ontologienamen, Dateinamen und primitiven Typen.

Teil II

Beschreiben von Ontologien mit DSD

Im Folgenden werden die einzelnen Beschreibungselemente der DSD genau beschrieben und an Beispielen erläutert. Dabei werden die drei Repräsentationsformen f-dsd, g-dsd und j-dsd parallel eingeführt. Auf die Erläuterung der Grammatik für f-dsd soll verzichtet werden.[3]

2 Definieren von Schemata

Zur Definition von Schemata stehen im Wesentlichen zwei Elemente zur Verfügung: Vordefinierte Datentypen (auch eingebaute oder primitive Datentypen) sowie selbstdefinierbare Datentypen (auch komplexe Datentypen oder Klassen).

2.1 Primitive Datentypen

2.1.1 Konzeption

Primitive Datentypen stellen Typen dar, die bereits vordefiniert sind. Durch sie werden allgemein gültige Sachverhalte wie Zahlen, Daten, Wahrheitswerte, Zeichenketten etc. abgebildet. Jeder dieser Typen besteht aus einer Menge gültiger Werte, sein Wertebereich, sowie eine oder mehrere lexikalische Repräsentationen dieser Werte. Diese Repräsentation kann je nach Anwendungsgebiet und DSD-Repräsentation wechseln.

Von DSD werden zur Zeit acht primitive Datentypen unterstützt. Ihre Definition und Repräsentation in f-dsd leitet sich von den in XML Schema [19] definierten primitiven Datentypen ab:

- `Integer`. Repräsentiert die Menge der ganzen Zahlen (positiv, null und negativ).
- `Double`. Repräsentiert die Menge der Fließkommazahlen (positiv, null und negativ).
- `String`. Repräsentiert die Menge der Zeichenketten, d.h. endliche Ketten von Einzelzeichen.
- `Boolean`. Repräsentiert die Menge der Wahrheitswerte, d.h. die Menge bestehend aus `wahr` und `falsch`.
- `Date`. Repräsentiert die Menge der Datumswerte. Ein Datum steht für einen vergangenen, jetzigen oder zukünftigen Tag in der Zeit, also ein nicht-periodisches Ereignis. Ein Tag beginnt um 0:00 Uhr und endet mit dem Beginn des folgenden Tages.
- `Time`. Repräsentiert die Menge der Zeitpunkte an einem Tag. Eine solcher Zeitpunkt wiederholt sich an jedem Tag. Seine Dauer ist null.
- `DateTime`. Repräsentiert die Menge der einzelnen Punkte in der Zeit. Die Dauer eines Zeitpunkts ist null. `DateTime` kann als Kombination von `Date` und `Time` angesehen werden.
- `Duration`. Repräsentiert die Menge der Zeitdauern. Eine Zeitdauer hat keinen bestimmten Anfangs- und Endzeitpunkt, sondern steht stellvertretend für jedes Zeitintervall dieser Länge.

[3] Die Grammatik kann unter `http://www.ipd.uka.de/DIANE/docs/f-dsd-pure.g` heruntergeladen werden.

2.1.2 Repräsentationen

Die Syntax zur Notation der primitiven Datentypen ist in den drei DSD-Repräsentationsformen g-dsd, f-dsd und j-dsd unterschiedlich. In g-dsd wird ein primitiver Datentyp durch ein weißes Kästchen dargestellt, das den Namen des Typs in fetter Schreibmaschinenschrift trägt (siehe Abbildung 4).

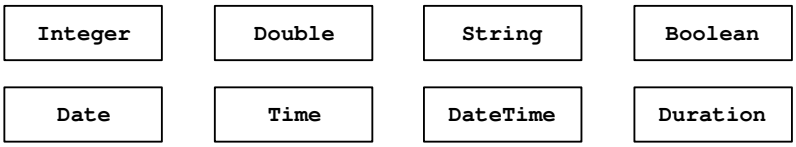


Abbildung 4: Primitive Datentypen in g-dsd.

In f-dsd erfolgt die Notation durch den Namen des primitiven Typs in Schreibmaschinenschrift: `Integer`, `Double`, `String` etc.

In j-dsd werden die primitiven Typen nicht direkt durch die in Java eingebauten Datentypen `int`, `double` etc. abgebildet, sondern durch spezielle Klassen nachgebildet. Diese Klassen haben den Namen des primitiven Datentyps mit einem vorangestellten `XSD_`, um sie von gleich lautenden Java-Klassen unterscheiden zu können. Alle diese Klassen erben von der abstrakten Oberklasse `XSD_Type`, was sie als primitive Typen kennzeichnet. `XSD_Type` erbt von der allgemeinen Oberklasse `Thing` (siehe Abschnitt 2.2.1). Die Klassen befinden sich im Javapaket `dsd.elements.xsd`. Intern werden die Werte entweder durch die eingebauten Javatypen repräsentiert (`int, double, boolean`), durch analoge Javaklassen abgebildet (`String`) oder in anderen Datenstrukturen abgespeichert (`java.util.GregorianCalendar` oder `long` für die Datumstypen). Zur Erzeugung von primitiven Werten steht daher ein entsprechender Konstruktor zur Verfügung. Eine beispielhafte Definition von `XSD_Integer` sieht wie folgt aus:

```
package dsd.elements.xsd;

public abstract class XSD_Type extends Thing {}

public class XSD_Integer extends XSD_Type
{
   private int value;

   XSD_Integer(int value)
   {
       this.value = value;
   }
}
```

2.1.3 Ordnungsrelationen

Auf jedem primitiven Datentyp ist eine totale Ordnungsrelation definiert, mit deren Hilfe je zwei Werte des selben Typs verglichen werden können. Für die zahlen- und zeitbezogenen Typen ist die Relation klassisch definiert, `String`s werden lexikographisch verglichen, für `Boolean` gilt, dass `falsch` kleiner als `wahr` ist. Die Ordnungsrelationen ermöglichen es, folgende Vergleiche auf dem Wertebereich durchzuführen: $==, <=, >=, <, >, ! =$. In f-dsd und g-dsd werden diese genau mit diesen Vergleichsoperatoren notiert. In j-dsd steht dazu in `XSD_Type` die abstrakte Methode `int`

`compareTo(XSD_Type value)` zur Verfügung, welche in den konkreten Untertypen implementiert wird. Sie liefert -1, 0 oder 1, je nachdem ob der Referenzwert kleiner, gleich oder größer als der Vergleichswert `value` ist.

2.1.4 Unscharfe Ordnungsrelationen

Neben der scharfen Ordnungsrelation sind noch unscharfe (engl. fuzzy) Ordnungsrelationen definiert, die einem Wertepaar nicht binär vergleichen, sondern noch gewisse Toleranzen zulassen. Zwar sind die Zahlen 100 und 100.1 nicht exakt gleich, in manchen Fällen ist es jedoch sinnvoll festzuhalten, dass sie *fast* gleich sind und die Ähnlichkeit mit einem Fließkommawert auszudrücken. Hierzu wird in j-dsd eine zusätzliche Methode `double fCompareTo(XSD_Type value)`[4] eingeführt, die einen Fließkommawert aus [-1,1] zurück gibt. Für die einzelnen konkreten Datentypen ist sie wie folgt definiert: Zahlenbezogene Werte dürfen um bis zu 10% nach oben oder unten vom Referenzwert abweichen, erst danach gelten sie als wirklich kleiner (+1) oder wirklich größer (-1). Dazwischen ändert sich der Rückgabewert kontinuierlich und linear von +1.0 über 0.0 zu -1.0. Abbildung 5 veranschaulicht diese Beziehung für den beispielhaften Referenzwert 100.

Für datumsbezogene Werte gelten folgende Festlegungen: Für `Date` und `DateTime` ist keine standardmäßige Abweichung definiert. Für `Time` ist eine Abweichung von +/- 10% einer Tageslänge, für Duration eine Abweichung von +/- 10% von der Referenzdauer zugelassen. Für `String` und `Boolean` ist keine unscharfe Vergleichsfunktion definiert; hier entspricht `fCompareTo` der scharfen `compareTo`-Methode.

Mit Hilfe der Methode `fCompareTo` können nun die bekannten Vergleichsoperatoren so geändert werden, dass sie nicht nur streng gelten (1) oder nicht gelten (0), sondern auch den Zwischenbereich von 0.0 und 1.0 annehmen können. Die Notation erfolgt durch ein vorangestelltes $\sim$:

- Berechne $a \sim == b$ durch $1 - |$`a.fCompare(b)`$|$
- Berechne $a \sim <= b$ durch `a.fCompare(b) >= 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim < b$ durch `a.fCompare(b) > 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim >= b$ durch `a.fCompare(b) <= 0 ? 1.0 : a.fCompare(b)`
- Berechne $a \sim > b$ durch `a.fCompare(b) < 0 ? 1.0 : a.fCompare(b)`

[4] Das `f` steht für fuzzy, deutsch also unscharf.

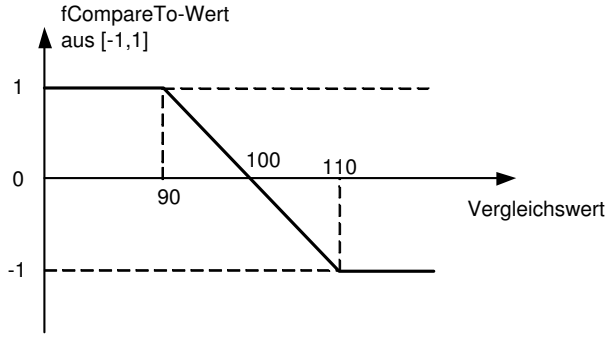


Abbildung 5: Unscharfer Vergleich des Referenzwertes 100.

- Berechne $a \sim! = b$ durch |`a.fCompare(b)`|

Neben den vordefinierten unscharfen Vergleichsoperatoren, die eine 10%-ige Abweichung zulassen, kann die erlaubte Abweichung auch selbst definiert werden. Hierzu wird in f-dsd und g-dsd in eckigen Klammer die minimal und/oder maximal erlaubte prozentuale oder absolute Abweichungsgrenze angegeben[5]. Ein Beispiel könnte sein $x \sim< [20\%]y$ oder $100 \sim== [75, 125]y$. Im ersten Fall darf der Vergleichswert y auch noch um bis zu 20% größer sein als der Referenzwert x um noch als *ungefähr kleiner* zu gelten, im zweiten Fall gilt der Wert von y noch als *ungefähr gleich* zu 100, falls er im Bereich zwischen 75 und 125 liegt. Der Ähnlichkeitswert fällt jeweils linear. In j-dsd können diese Angaben als weitere Parameter der Methode `fCompareTo` gemacht werden.

2.2 Klassen

Im Folgenden soll erläutert werden, wie Typen in Form von Klassen selbst definiert werden können.

2.2.1 Konzeption

Klassen sind Datentypen, die nicht von DSD fest vorgegeben werden, sondern von der Community selbst definiert werden können. Ihre Anzahl ist daher prinzipiell nicht beschränkt. Klassen stehen stellvertretend für die Sammlung aller Individuen der realen Welt einer bestimmten Art. Beispielsweise steht die Klasse Movie für die Sammlung aller real existierender Filme, die Klasse Price für alle Preise. Klassen werden in g-dsd durch ein weißes Kästchen mit dem Klassennamen in fetter Normalschrift dargestellt, wie Abbildung 6 beispielhaft zeigt. In f-dsd wird einfach der Klassenname gefolgt von der Attributdefinition (siehe Abschnitt 2.2.2) in eckigen Klammer notiert:

```
Movie[]
Price[]
```

Klassennamen beginnen stets mit einem Großbuchstaben.

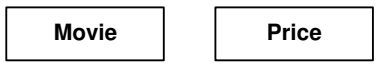


Abbildung 6: Beispielhafte Klassendefinition in g-dsd.

In j-dsd wird das Konzept der Klasse auf gewöhnliche öffentliche Javaklassen abgebildet. Hier würden die Klassen Movie und Price als

```
public class Movie extends Thing {}
public class Price extends Thing {}
```

dargestellt und wie in Java üblich in einzelnen Dateien `Movie.java` und `Price.java` abgelegt. Jede Klasse erbt von der allgemeinen Oberklasse `Thing` aus dem Paket `dsd.elements` (falls nicht anders angegeben, siehe Abschnitt 2.2.5).

[5] Diese Möglichkeit ist noch nicht in f-dsd implementiert

2.2.2 Attribute

Haben die durch die Klasse vertretenen Individuen gemeinsame Eigenschaften, so können diese durch Attribute repräsentiert werden. Ein Attribut hat einen Namen und einen Zieltyp, d.h. einen Typ der angibt, durch welche Werte/Instanzen das Attribut „gefüllt“ werden kann. Man sagt auch, das Attribut sei von diesem Typ. Der Zieltyp kann sowohl ein primitiver Datentyp als auch eine Klasse sein. Im Beispiel könnte die Klasse Movie die Attribute title vom Typ `String`, length vom Typ `Duration` und mainActor vom Typ Actor besitzen. Der Name eines Attributes beginnt immer mit einem Kleinbuchstaben. Auf Instanzebene nennt man den Wert, der ein Attribut ausfüllt, *Füllwert* oder englisch Filler.

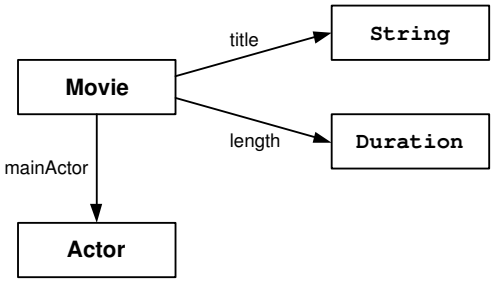


Abbildung 7: Beispielhafte Attributdefinition in g-dsd.

In g-dsd werden Attribute durch Pfeile dargestellt, die von der Klasse zum Zieltyp zeigen und mit dem Namen des Attributes beschriftet sind. Abbildung 7 zeigt das am Beispiel.

In f-dsd erfolgt die Definition der Attribute in den eckigen Klammern nach der Klassendefinition, jeweils als <`Attributname`> : <`Zieltyp`> durch Komma getrennt. Im Beispiel wäre das:

```
Movie
[
   name: String,
   length: Duration,
   mainActor: Actor
]
```

In j-dsd werden Attribute durch gewöhnliche öffentliche Javaattribute der zugehörigen Javaklasse dargestellt. Ihr Typ entspricht dem Zieltyp. Im Beispiel wäre das:

```
public class Movie extends Thing
{
   public XSD_String name;
   public XSD_Duration length;
   public Actor mainActor;
}
```

Grundsätzlich sind alle Attribute optional und einwertig, d.h. in UML hätten sie die Kardinalität 0..1. Durch die Verwendung listenwertiger bzw. definierender Attribute kann dies geändert werden. Diese werden im nächsten Abschnitt vorgestellt.

2.2.3 Listenwertige Attribute

Ist ein Attribut so beschaffen, dass es nicht nur einen, sondern eine geordnete Liste an Elementen des Zieltyps aufnehmen können muss, so wird dies durch ein listenwertiges Attribut dargestellt. Beispielsweise hat eine Route ein listenwertiges Attribut passes mit dem Zieltyp Town, welches eine geordnete Reihe an Städten erfassen kann, welche durch diese Route durchfahren werden.

In g-dsd notiert man listenwertige Attribute durch die Angabe von `{list}` an der Pfeilspitze (siehe Abbildung 8).

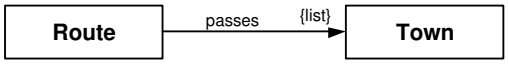


Abbildung 8: Beispielhafte listenwertige Attributdefinition in g-dsd.

In der Syntax von f-dsd erfolgt die Markierung als listenwertiges Attribute durch Setzen von `list of` vor der Angabe des Zieltyps:

```
Route
[
   passes: list of Town
]
```

Die Darstellung in j-dsd ist schwieriger, da Java das Konzept mengen- oder listenwertiger Attribute direkt nicht kennt. Aus diesem Grund wurde die zusätzliche Klasse `ThingList` eingeführt, welche eine geordnete Liste an Elementen eines bestimmten, angegebenen Typs aufnehmen kann. `ThingList` befindet sich wie alle Hilfskonstrukte im Javapaket `dsd.elements`. Intern verwaltet sie ein Attribut `base` vom Typ `Thing`, welches eine Beispielinstanz der Klasse der zu speichernden Objekte aufnimmt, sowie einen Java-Vector `data` zur Aufnahme der eigentlichen Objekte. Auch `ThingList` erbt von `Thing`. Das Beispiel sieht dann wie folgt aus:

```
public class ThingList extends Thing
{
   public Thing base;
   private java.util.Vector data;
}

public class Route extends Thing
{
   ThingList passes = new ThingList();

   public Route()
   {
       passes.base = new Town();
   }
}
```

Bereits hier wird ersichtlich, dass j-dsd durch Java-bedingte Kunstgriffe für den Menschen wesentlich unübersichtlicher ist als f-dsd und daher nur für die interne Verarbeitung verwendet werden sollte.

In UML hätten listenwertige Attribute die Kardinalität 0..*.

2.2.4 Attributarten

Attribute werden in DSD in drei Arten unterschieden: definierende (engl. definitional) , ableitbare (engl. incidental) und orthogonale Attribute. Dabei gilt folgender Grundsatz: Sind von einer Instanz die Füllwerte aller definierenden Attribute bekannt, so sind dadurch auch die Füllwerte der ableitbaren Attribute bestimmt. Anders ausgedrückt sind zwei Instanzen mit definierenden Attributen gleich, wenn sie gleiche Füllwerte in ihren definierenden Attributen besitzen. In der Welt der Datenbanken findet sich mit dem Konzept des Schlüssels ein ähnliches Konzept. In der Beispielklasse Movie wäre das Attribut name definierend, während length und mainActor ableitbar sind (in der Annahme, dass es keine zwei gleich heißenden Filme gibt). Orthogonale Attribute sind nicht durch die definierenden Attribute bestimmt. Sie stellen nicht-inhärente Eigenschaften dar, die der Instanz zugewiesen werden. Zwei gleiche Instanzen können sich daher in ihren orthogonalen Attributen unterscheiden.

In g-dsd werden definierende Attribute durch einen schwarz ausgefüllten Kreis am Pfeilende markiert. Ableitbare Attribute bleiben ohne Markierung (siehe Abbildung 9). Orthogonale Attribut werden durch einen unausgefüllten Kreis markiert.

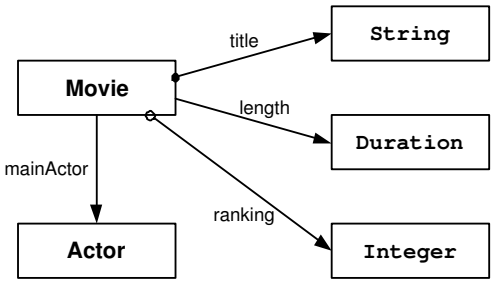


Abbildung 9: Unterscheidung von definierenden und ableitbaren Attributen in g-dsd.

In f-dsd werden definierende Attribute mit dem vorangestellen Schlüsselwort `defprop` markiert, ableitbare bleiben auch hier ohne Markierung, orthogonale mit dem Schlüsselwort `orthprop`:

```
Movie
[
    defprop name: String,
    length: Duration,
    mainActor: Actor,
    orthprop ranking: Integer
]
```

In j-dsd ist die Markierung schwieriger und geschieht durch die statischen Vektoren `defprops` und `orthprops` in `Thing`. In ihnen wird jedes definierende bzw. orthogonale Attribut mit seinem vollständigen Namen eingetragen. Dies geschieht in einem statischen Initialisierer der Klasse des definierenden Attributs. Im Beispiel:

```
public class Movie extends Thing
{
   public XSD_String title;
   static {defprops.add("domain.movie.Movie.title");}

   public XSD_Duration length;
```

```
    public Actor actor;

    public XSD_Integer ranking;
    static {orthprops.add("domain.movie.Movie.ranking");}
}
```

2.2.5 Vererbung

Eine besondere Art der Beziehung zwischen Klassen stellt die Vererbung dar. Sie drückt eine *is-a*-Beziehung aus. Eine Klasse K sollte von einer Klasse L erben, wenn L alle Elemente enthält, die auch K enthält, K also eine Spezialisierung von L ist. Es gilt, dass K alle Attribute, die in L definiert sind, übernimmt und zusätzliche weitere definieren kann. Beispielsweise sollte die Klasse Actor von Person erben, da jeder Schauspieler eine Person ist. Attribute von Person, wie name, birthdate etc. werden dann übernommen. In DSD kann jede Klasse nur von maximal einer Oberklasse erben, d.h. das Konzept der Mehrfachvererbung existiert nicht. Erbt eine Klasse (außer Thing) von keiner Klasse, so erbt sie implizit von Thing.

In g-dsd wird die Vererbung durch den aus UML bekannten unausgefüllten Pfeil dargestellt, der von der Unter- zur Oberklasse zeigt. Ein Beispiel zeigt Abbildung 10.

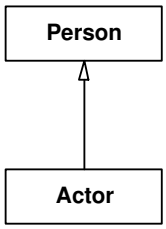


Abbildung 10: Beispiel für eine Vererbungsbeziehung in g-dsd.

In f-dsd wird die Vererbung durch `extends <Name der Oberklasse>` nach dem Namen der Klasse notiert:

```
Actor extends Person []
```

In j-dsd wird der Mechanismus aus Java genutzt, um eine Vererbung darzustellen. Ein Erben von Thing muss hier explizit angegeben werden:

```
public class Person extends Thing {}

public class Actor extends Person {}
```

Wichtig: Für die Vererbung gelten Einschränkungen, wenn die Klassen spezielle Markierungen haben. Siehe dazu Abschnitt 3.5.

2.3 Ontologien

Um den Raum der Klassen übersichtlich zu halten, ist es wichtig, diesen zu strukturieren. Hierzu wird in DSD das Konzept der Ontologien verwendet. Eine Ontologie ist eine Sammlung von Klassen (und Instanzen, siehe später), die thematisch zusammengehören.

```
{ontology domain.movie}

Movie
[
   defprop title: String,
   length: Duration,
   mainActor: Actor
]

Actor
[
   name: String
]
```

Abbildung 11: Datei `domain.movie.dsd` zur Definition der Ontology `domain.movie` in f-dsd.

2.3.1 Ontologienamen

Ontologien werden durch einen eindeutigen Namen unterschieden. Generell unterscheidet man Ontologien auf den drei durch DSD vorgegebenen Ebenen, was auch den ersten Bestandteil dieses Namens bestimmt: Die obere Dienstontologie (bezeichnet durch `upper`), Kategorieontologien (bezeichnet durch `category`) und Domänenontologien (bezeichnet durch `domain`). In diesen Gruppen können hierarchische Untergruppen eröffnet werden, die jeweils thematisch abgeschlossen sind. Ihre Namen werden analog zum Namensraum-Mechanismus aus XML oder dem Package-Mechanismus aus Java durch Punkte strukturiert. Start ist immer der Bezeichner der Hauptgruppe. Um beispielsweise die Domäne der Filmwelt zu beschreiben, könnte die Ontologie `domain.cinema` verwendet werden, die Welt der Fahrzeuge könnte man in `domain.transportation.vehicles` modellieren. Die Kategorie der Realweltdienste findet sich in `category.realobjectservice`, die Grundkonzepte zum Dienstprofil in `upper.profile`. Zu bemerken ist, dass durch die Punktnotation keine Inklusion definiert wird: Konzepte aus `a.x.y` sind nicht automatisch auch in `a` und `a.x` enthalten.

2.3.2 Neuerstellung von Ontologien

Die Sammlung der Klassen und Instanzen einer zu definierenden Ontologie werden in f-dsd in einer Datei zusammengefasst. Diese beginnt mit dem Markierer `{ontology <Name>}`, mit dem die Neudefinition der Ontologie mit dem angegebenen Namen eingeleitet wird. In der Datei selbst können eine beliebige Anzahl von Klassen und Instanzen definiert werden, welche alle in der angegebenen Ontologie abgelegt werden. Zwar ist eine Mischung von Instanzen und Klassen erlaubt, dennoch sollte sich in einer Datei aus Gründen der Übersichtlichkeit auf eine Art beschränkt werden. Der Name der Datei ist dann der Name der Ontologie mit der Endung `.dsd` für DIANE Schema-Definitionen und `.did` für DIANE Instanz-Definitionen. Abbildung 11 zeigt einen Ausschnitt aus der Datei `domain.movie.dsd`, welche das Domäne der Filme repräsentiert.

Wichtig ist, dass generell alle in einer Ontologiedefinition verwendeten Typen entweder primitiv sind oder selbst in dieser Ontologie definiert werden. Klassen aus anderen Ontologien müssen gekennzeichnet werden (siehe Abschnitt 2.3.3).

In g-dsd erfolgt die Kennzeichnung einer Neudefinition von Ontologien ähnlich. Hier werden Ontologien auf einem Zeichenblatt zusammengefasst, welches in der linken oberen Ecke mit dem Schlüsselwort ONTOLOGY: <Name: > markiert wird. Auch hier gilt, dass alle zur Definition verwendeten Typen entweder primitiv sind oder innerhalb derselben Ontologie definiert wurden. Extern definierte Klassen müssen gekennzeichnet werden (siehe 2.3.3).

Für die Umsetzung des Ontologiekonzeptes in j-dsd wird das vorhandene Java-Package-Konzept verwendet. Dabei gilt, dass eine Klasse der Ontologie `name` im Javapaket `dsd.schema.name` zum Liegen kommt. Dies führt dazu, dass die Pakete der primitiven Datentypen sowie die der speziellen Konstrukte wie `Thing` oder `ThingList` explizit über `import` eingebunden werden müssen. Klassen derselben Ontologie liegen im gleichen Javapaket und können somit direkt verwendet werden. Folgender Code zeigt die Definition von Movie in j-dsd:

```
package dsd.schema.domain.movie;

import dsd.elements.*;
import dsd.elements.xsd.*;

public class Movie
{
   public XSD_String title;
   static {defprops.add("domain.movie.Movie.title");}

   public XSD_Duration length;
   public Actor actor;
}
```

2.3.3 Verwenden vorhandener Ontologien

Die Wiederverwendung von Klassen und Instanzen aus bereits existierenden Ontologien ist nicht nur eine Möglichkeit, den Aufwand einer Neuerstellung zu sparen, sondern verpflichtend (siehe Abschnitt 2.4). Bei der Einbindung solcher externer Klassen in die eigene Ontologie muss die ontologische Herkunft der Klasse angegeben werden.

In g-dsd erfolgt das durch Setzen der bereits extern definierten Klasse als Kästchen mit gestrichelter Umrandung. Zudem steht der Name der Herkunftsontologie in kleiner, kursiver Normalschrift oberhalb des Klassennamens. Abbildung 12 zeigt, wie die Klasse Person aus der externen Ontologie `domain.person` verwendet wird, um in der Ontologie `domain.movie` die Klasse Actor zu definieren.

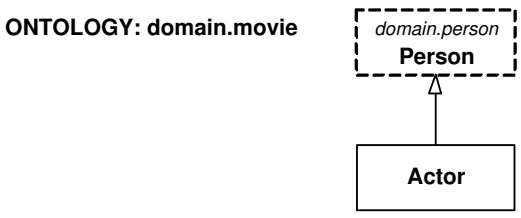


Abbildung 12: Verwendung der in `domain.person` definierten Klasse Person zur Neudefinition von Actor in `domain.movie`

In f-dsd geschieht die Markierung durch die Angabe von `at <Ontologiename>` nach der externen Klasse. Eine beispielhafte Definition von Actor, Person und Movie könnte daher wie folgt aussehen:

```
{ontology domain.person}
Person
[
   name: String,
   address: Address at domain.location
]

{ontology domain.movie}
Actor extends Person at domain.person
[
   inContractWith: Studio at domain.companies.entertainment,
   awards: list of Award
]

Movie
[
   defprop title: String,
   length: Duration,
   mainActor: Actor,
   productionCosts: Price at domain.money,
   filmLocation: Location at domain.location
]
```

Da j-dsd den gewöhnlichen Package-Mechanismus von Java nutzt, um Ontologien abzubilden, kann die Referenzierung von Klassen aus anderen Ontologien durch zwei bekannten Mechanismen erfolgen: (1) Einbinden der benötigten Klassen über `import`-Anweisungen oder (2) vollständige Benamung der Klassen mittels der Punkt-Notation. Folgendes Beispiel nutzt beide Möglichkeiten:

```
package dsd.schema.domain.movie;
import dsd.schema.domain.companies.entertainment.Studio;

public class Actor extends dsd.schema.domain.person.Person
{
   public Studio inContractWith;
}
```

2.4 Forderung: Einigung auf Klassen und Ontologien

Wie bereits in der Einleitung angeklungen, gilt für Schemata (wie auch Instanzen und Ontologien) der Grundsatz, dass diese als Ergebnis eines Einigungsprozesses in der Community erstanden sind. Daraus folgt die Forderung, dass keine zwei Klassen existieren, welche für die selben Individuen der realen Welt stehen. Beispielsweise ist es unzulässig, gleichzeitig die Klassen Movie und Film zu besitzen, die dasselbe bedeuten, aber andere Namen und evtl. auch andere Attribute und Oberklassen besitzen. In einem solchen Fall fehlt die angesprochene Einigung in der Community. Diese müsste sich für eine Klasse entscheiden und in Zukunft ausschließlich diese nutzen.

Konsequenterweise erstreckt sich diese Forderung auch auf ganze Ontologien. Ontologien müssen disjunkt sein, d.h. keine Klasse (oder Instanz) darf in mehr als einer Ontologie definiert werden.

In der Praxis erweist sich die Durchsetzung dieser Forderungen als nicht immer einfach: Einerseits kann die Community sehr groß werden, was eine direkte Einigung erschwert oder unmöglich macht,

andererseits kann die Community sehr dynamisch sein, etwa in einem mobilen Netz, wo ständig Teilnehmer der Community beitreten oder diese verlassen. In solchen Fällen ist es üblich, in gewissen Maßen die Forderung abzuschwächen und „persönliche Ontologien“ zu erlauben. Diese müssen jedoch vor der Verwendung in DSD durch Verfahren des Ontologie-Matchings (semi)automatisch aufeinander abgebildet werden, um eine Vereinheitlichung zu erlangen.

2.5 Verwandte Sprachen zur Schemadefinition

Ziel von DSD war nicht die Schaffung einer vollkommen neuartigen Syntax zur Beschreibung von Diensten. Gerade im Bereich der Schemabeschreibung existieren bereits viele Ansätze, an die DSD angelehnt wurde. Für f-dsd sind dies im Wesentlichen objektorientierte Programmiersprachen wie Java oder C++. Die Konzepte von Attributen und Füllwerten stammen aus F-Logic [6]. Die primitiven Datentypen sind den XML Schema Datatypes entliehen [19]. Die saubere Trennung zwischen Schema und Instanz in T-Box und A-Box findet sich vorwiegend in der Beschreibungslogik [1]. Das Konzept der Ontologiebenennung ist vom Packagekonzept in Java sowie den XML-Namespaces beeinflusst. g-dsd ist im Wesentlichen durch die Notation von Klassendiagrammen in UML geprägt.

2.6 Tipps zur Definition guter Schemata für Dienstbeschreibungen

Die Modellierung eines Schemas ist ein schwieriger Prozess, da dem Entwickler viele Freiheiten zur Verfügung stehen. Eine gute Lösung hängt von der späteren Verwendung ab. Schemata, die in DSD definiert werden, sind in der Regel speziell für die Instanziierung innerhalb von Dienstbeschreibungen gedacht. Im Folgenden sollen daher einige Tipps und Hinweise gegeben werden, wie Schemata in DSD definiert werden sollten.[6]

- **Überprüfe vorhandene Ontologien.** Bei der Entwicklung sollten ständig die bereits existierenden Ontologien und deren Klassen im Auge behalten werden. Laut der Forderungen aus Abschnitt 2.4 dürfen diese nicht erneut definiert werden, sondern müssen integriert oder erweitert werden.
- **Top-Down-Vorgehensweise.** Als generelle Vorgehensweise hat sich eine Modellierung von grob nach fein als günstig erwiesen. Beginnen sollte man dabei mit der Notation der fundamentalen Klassen. Diese sollten notiert werden, ohne ihre Attribute mit primitiven Zieltypen aufzulisten. Die Klassen sollten dann durch Vererbungsbeziehungen Generalisierung und durch Attribute verbunden werden. Erst danach sollten die primitiven Attribute hinzugefügt werden.
- **Nicht in Diensten denken.** Bei der Erstellung der Domänenontologie sollte man nicht ständig an einen konkreten Dienst denken, der damit beschrieben werden soll. Der Fokus sollte darauf liegen, die Domäne eigenständig zu beschreiben. Insbesondere sollte darauf geachtet werden, dass keine dienstspezifischen Eigenschaften (wie Effekte, Eingaben, Ausgaben) in die Beschreibung integriert werden. Auch Mengen und Variablen (siehe Abschnitte 4 und 5) dürfen in der Schemadefinition nicht auftauchen. Generell sollte man sich die Frage stellen: Was könnte einen Anfrager an der Domäne interessieren (auch wenn das evtl. über die Fähigkeiten eines bestimmten Dienstangebots hinaus geht)?
- **Primitive Typen nicht beschreiben.** Primitive Typen (insbesondere die datumsbezogenen Typen) brauchen nicht als neue Klassen modelliert zu werden. Auch Klassen für Wert-Einheit-Paare (wie z.B. Price oder WeightMeasure) sind üblicherweise schon vorhanden.

[6] Diese wurden unter anderem bei einer DIANE-Klasurtagung erarbeitet, bei der Studierende eigenständig Schemata zur Dienstbeschreibung entwickeln und ihre Vorgehensweise und Fehlentscheidungen protokollieren sollten.

- **Nicht in Tabellen denken.** Die Modellierung ähnelt zwar dem Entwurf eines relationalen Schemas, dennoch sollte man sich nicht verleiten lassen, in Tabellen zu denken. Die hierin vorkommenden Konzepte wie künstliche Schlüssel oder Fremdschlüssel sind daher zu vermeiden.
- **Unstrukturierte, dokumentartige Attribute vermeiden.** Attribute, welche Informationen in Form von unstrukturierten Dokumenten sammeln, sollten vermieden werden. Beispiele hierfür sind `textDescription` bei Movie oder `additionalInfo` bei Cinema. Häufig ist deren Typ ein „langer“ `String`. Diese sind für eine menschliche Verarbeitung bestimmt und können von Rechnern nicht sinnvoll ausgewertet werden. Für eine automatische Dienstnutzung sind sie daher ungeeignet und sollten entsprechend strukturiert oder weggelassen werden.
- **Listenwertige Attribute sparsam verwenden.** Speziell bei der Definition von Schemata, die später als Grundlage für Dienstbeschreibungen dienen, sollte darauf geachtet werden, listenwertige Attribute möglichst sparsam einzusetzen. Das Problem bei listenwertigen Attributen liegt im Vergleicher. Dieser muss aus Gründen der Berechenbarkeit bestimmt Anforderungen an solche Attribute stellen. Zur Umgehung dieser Attribute kann man ausnutzen, dass solche Attribute häufig auch dem Zieltyp zugeschlagen werden können und dort dann nicht mehr listenwertig sind. Generell sollte man Attribute daher von den Teilen zum Ganzen modellieren, z.B. ein Haus liegt in einer Straße, welche in einer Stadt liegt, welche in einem Land liegt etc. anstatt umgekehrt.
- **Redundante Attribute vermeiden.** Attribute, welche bereits durch andere Pfade abgeleitet werden können, sollten nicht in die Ontologie aufgenommen werden, da sie später zu Problemen beim Vergleich führen können. Eine Straße liegt natürlich auch in einem Land, dies kann jedoch über das Attribut der Zwischenklasse Stadt abgelesen werden.

3 Definieren von Instanzen

Instanzen sind das zweite grundsätzliche Modellierungselement von DSD und besonders im Bereich von Dienstbeschreibungen sehr wichtig. Unterschieden werden generell Instanzen primitiver Typen (primitive Werte) und Instanzen von Klassen.

3.1 Werte primitiver Typen

Werte primitiver Typen (auch primitive Werte) sind Elemente aus dem Wertebereich primitiver Datentypen. Die Elemente bilden keine Einheiten im objektorientierten Sinne, sondern ihre Gleichheit ist durch eine spezielle Funktion definiert, welche ausschließlich die Struktur des Werts in Betracht zieht. Für jeden primitiven Wert existiert zumindest eine lexikalische Hauptrepräsentation. Wir nennen diese Standardrepräsentation für primitive Werte. Für die einzelnen primitiven Datentypen sieht sie wie folgt aus:

- `Integer.` Werte sind die ganze Zahlen.
  Syntax: `(PLUS|MINUS)? ZIFFER+`
  Beispiele: `-4, 0, 42`
- `Double.` Werte sind Fließkommazahlen.
  Syntax: `(PLUS|MINUS)? ZIFFER+ PUNKT ZIFFER+`
  Beispiele: `-0.22, 0.0, 42.1`
- `String.` Werte sind endliche Zeichenketten, die durch doppelte Anführungszeichen terminiert sind.
  Syntax: `ANFUEHUNGSZEICHEN ZEICHEN* ANFUEHRUNGSZEICHEN`
  Beispiel: `"Das ist ein TestString"`
- `Boolean.` Werte sind die beiden Wahrheitswerte in englischer Sprache und in spitzen Klammern.
  Syntax: <`true`> | <`false`>
- `Date.` Werte sind Datumsangaben im Format <YYYY-MM-DD>.
  Syntax: < `ZIFFER ZIFFER ZIFFER ZIFFER MINUS ZIFFER ZIFFER MINUS ZIFFER ZIFFER` >
  Beispiele: <`1976-03-12`>, <`2005-01-01`>
- `Time.` Werte sind Zeitangaben im Format <HH:MM:SS.sss>.
  Syntax: < `ZIFFER ZIFFER DOPPELPUNKT ZIFFER ZIFFER`
  `(DOPPELPUNKT ZIFFER ZIFFER (PUNKT ZIFFER ZIFFER ZIFFER)?)?` >
  Beispiele: <`10:15`>, <`20:15:10.43`>
- `DateTime.` Werte sind Zeitpunktangaben im Format bestehend aus einem `Date`- und einem `Time`-Werte (ohne spitze Klammern), welche durch ein `T` getrennt sind und zusammen in spitze Klammern gesetzt werden.
  Beispiel: <`1976-03-12T10:15:10`>
- `Duration.` Werte sind Zeitlängenangaben im Format <PyYm_1MdDThHm_2MsS>. Hierbei steht y für die Anzahl der Jahre, m_1 für die Anzahl der Monate, d für die Anzahl der Tage, h für die Anzahl der Stunden, m_2 für die Anzahl der Minuten, s für die Anzahl der Sekunden und Bruchteile von Sekunden. s ist eine Fließkommazahl, alle anderen sind Ganzzahlen. Werte, die 0 sind, können zusammen mit ihrem großbuchstabigen Markierer ausgelassen werden. Alle Werte beginnen mit dem Startsymbol `P` (für engl. period) und haben das Trennzeichen `T`.

Beispiele: <P1M10DT> für 1 Monate und 10 Tage, <PT10M> für 10 Minuten, <PT1M10.5S> für 1 Minute, 10 Sekunden und 500 Millisekunden.

In f-dsd werden Werte primitiver Typen in der oben vorgestellten Standardrepräsentation notiert. Auch g-dsd verwendet diese Syntax. Hier werden die Werte direkt auf die Zeichenfläche geschrieben, d.h. sie werden nicht wie die primitiven Typen in Kästchen notiert.

In j-dsd werden primitive Werte durch Javaobjekte der speziellen XSD-Typen dargestellt. Zur Erzeugung hat jeder XSD-Typ einen oder mehrere Konstruktoren (siehe auch Abschnitt 2.1.2), die als Parameter den Wert in einem eingebauten Javatyp (wie `int`, `double`, `boolean`), einem Objekttyp (wie `String` oder `GregorianCalendar`) oder als so genannten Valuestring erwarten, welcher im Wesentlichen der vorgestellten Standardrepräsentation entspricht. Für jeden Typ existiert eine geeignete `toString`-Methode, die den Wert in der Standardrepräsentation ausgibt.

3.2 Instanzen von Klassen

Instanzen sind ein Kernkonzept von DSD. Jede Instanz steht stellvertretend für ein Individuum der realen Welt und gehört eindeutig zu einer speziellsten Klasse K und implizit zu all ihren Oberklassen. Instanzen werden in zwei Arten eingeteilt: benamte und anonyme Instanzen.

- **Benamte Instanzen.** Benamte Instanzen stehen stellvertretend für reale Individuen, welche als Ganzes bzw. als Einheit angesehen werden. Sie haben einen global eindeutigen Namen, unter dem sie im *Instanzenpool* abgelegt werden können. Dieser ist für die gesamte Community zugänglich. Aus diesem Grund erstreckt sich der Einigungsprozess der Community neben den Schemata auch auf die benamten Instanzen im Pool. Wie alle Instanzen sind auch benamte Instanzen eindeutig von einer Klasse abgeleitet, welche auch die Ontologiezugehörigkeit bestimmt. Anfragen nach benamten Instanzen sind über deren Namen („genau die“) oder deklarativ über deren ausgefüllte Attribute („so eine“) möglich.
- **Anonyme Instanzen.** Anonyme Instanzen stehen stellvertretend für reale Individuum, die über ihre Attribute definiert sind und nicht als Einheit angesehen und wiederverwendet werden. Sie haben keinen Namen und werden auch nicht im Instanzenpool abgelegt. Anonyme Instanzen können daher zu jeder Zeit angelegt werden. Wie jede Instanz sind auch anonyme Instanzen eindeutig von einer Klasse abgeleitet. Anfragen nach anonymen Instanzen sind nur über deren ausgefüllte Attribute („so eine“) möglich.

3.2.1 Definition benamter Instanzen

In f-dsd können benamte Instanzen der Klasse k nur innerhalb der Ontologie definiert, welche auch die Klasse k enthält. Die Syntax lautet: `<Name der Instanz> as <Klassenname>`. Dabei muss der Name der Instanz *innerhalb der Klasse* eindeutig sein. Global wird er durch Anfügen des Ontologie- und Klassennamens vor dem Instanznamen eindeutig. Dateien, die Instanzdefinitionen enthalten, erhalten die Endung `.did`. Instanznamen beginnen immer mit einem Kleinbuchstaben. Im Beispiel werden in der Film-Ontologie Instanzen für einen bestimmten Film und einen bestimmten Schauspieler der realen Welt angelegt:

```
{ontology domain.movie}

harryPotter3 as Movie
danielRadcliffe as Actor
```

Auch in g-dsd können benamte Instanzen der Klasse k nur innerhalb der Ontologie definiert werden, welche die Klasse k enthält. Sie werden als weißes Kästchen dargestellt, welches den Namen der Instanz und die Klassennamen abgetrennt durch einen Doppelpunkt enthält. Es wird eine nicht-fette, unterstrichene Normalschrift verwendet. Abbildung 13 zeigt die Definition der Instanzen in g-dsd.

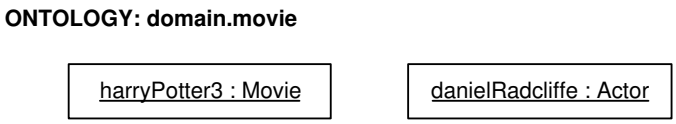


Abbildung 13: Definition zweier benamter Instanzen in g-dsd.

In j-dsd gestaltet sich die Definition benamter Instanzen schwieriger. Zwar werden Instanzen von DSD in j-dsd auf gewöhnliche Javaobjekte abgebildet, jedoch wird dadurch alleine keine Benamung und keine Dauerhaftigkeit der Objekte erreicht. Zur Behebung des Problems wird in j-dsd für jede benamte Instanz eine *Erzeugungsklasse* angelegt. Diese verfügt über eine statische Methode `create`, welche bei Bedarf das Javaobjekt zurückliefern kann. Der Name der Instanz wird durch das Attribut `instanceName` (geerbt von `Thing`) bestimmt, welches bei der Erzeugung gesetzt wird. Hierbei kommt der oben erwähnte, global eindeutige Name bestehend aus Ontologie-, Klassen- und Instanzname zum Einsatz. Die Erzeugungsklasse trägt den Namen der Instanz (mit beginnendem Kleinbuchstaben) und befindet sich im Javapackage, das sich aus Ontologie und Klassennamen ergibt und mit der Prefix `dsd.instance` beginnt. Die DSD-Klasse selbst muss über eine `import`-Anweisung eingebunden werden, da sie sich in einem anderen Package (`dsd.schema...`) befindet.

Eine Beispieldefinition einer Instanz sieht in j-dsd wie folgt aus:

```
package dsd.instance.domain.movie.Movie;

import dsd.schema.domain.movie.Movie;

public class harryPotter3
{
   public static Movie create()
   {
       Movie instance = new Movie();
       instance.instanceName = "dsd.instance.domain.movie.Movie.harryPotter3";
       return instance;
   }
}
```

Wichtig: Die Gleichheit benamter Instanzen wird in j-dsd nicht auf die Gleichheit von Javaobjekten (mittel `==`-Operator) zurückgeführt, sondern muss über die `equals`-Methode erfolgen. Diese testet die Gleichheit dann über einen Stringvergleich der zugehörigen `instanceName`s. Dies hat zwei Vorteile: Erstens muss die `create`-Methode nicht die in Umlauf gebrachten Javaobjekte kontrollieren, zweitens funktioniert der Ansatz auch in verteilten Umgebungen, in denen unterschiedliche Java Virtual Machines autonom ihre eigenen Javazeiger verwalten.

3.2.2 Definition anonymer Instanzen

Da anonyme Instanzen nicht unter einem Namen veröffentlicht werden, können sie überall definiert werden (z.B. auch während der Aufstellung einer Dienstbeschreibung). Sie besitzen keinen Namen,

jedoch einen eindeutigen Typ.

In f-dsd erfolgt die Definition durch `anonymous <Klassenname>`, wobei häufig die Angabe der Ontologie durch ein anschließendes `at <Ontologiename` nötig ist. Beispiel:

```
anonymous Price at domain.money
anonymous Address at domain.location
```

Die vorgestellten Definition machen so keinen Sinn, da sie erstens nicht über einen Namen aufgegriffen werden können und zweitens keine ausgefüllten Attribute besitzen (siehe nächster Abschnitt).

Die Definition einer anonymen Instanz in g-dsd ähnelt der Definition einer benamten Instanz, nur wird der Name der Instanz weggelassen. Abbildung 14 zeigt ein Beispiel.

Abbildung 14: Definition zweier anonymer Instanzen in g-dsd.

Da die Probleme von Benamung und Dauerhaftigkeit bei anonymen Instanzen entfallen, können diese in j-dsd sehr einfach durch Anlegen normaler Javaobjekte über den Konstruktor erstellt werden. Im Beispiel:

```
new dsd.schema.domain.money.Price();
new dsd.schema.domain.location.Address();
```

3.2.3 Füllen von Attributen

Instanzen sind eindeutig von einer Klasse abgeleitet, d.h. für sie ist eindeutig eine Menge von Attributen definiert, die ausgefüllt werden können. Attribute werden stets mit Instanzen oder Werten des Zieltyps oder eines Untertyps davon gefüllt. Diese werden als *Füllwert* bezeichnet. Nicht ausgefüllte Attribute sind möglich; sie gelten dann als unbekannt.

In f-dsd muss die Füllung der Attribute einer Instanz direkt nach der Definition der Instanz angegeben werden. Diese geschieht im *Instanzattributblock* , welcher durch eckige Klammern eingeschlossen wird. Darin erfolgt die Füllung in der Syntax `<Attributname> = <Füllwert>`. Mehrere Füllungen werden durch Komma getrennt. Als Füllwerte können (je nach Typ) primitive Werte, existierende benamte Instanzen oder an dieser Stelle definierte anonyme Instanzen dienen. Beispiel:

```
{ontology domain.movie}

harryPotter3 as Movie
[
   title = "Harry Potter und der Gefangene von Askaban",
   length = <PT2H21M>,
   mainActor = danielRadcliffe,
   productionCosts = anonymous Price
                     [
                        amount = 130000000,
                        currency = usd
                     ],
   filmLocation = greatBritain as Country at domain.location
]
```

Im Beispiel ist zu sehen, dass zwar benamte Instanzen und Klassen aus anderen Ontologien verwendet wurden (wie danielRadcliffe aus `domain.person` oder Price aus `domain.money`), deren Ontologienamen aber nicht explizit notiert wurden. Generell gilt, dass eine spezifizierende Typangabe weggelassen werden kann, wenn die benamte Instanz genau den Zieltyp des Attributs hat. Sie kann dann ja aus dem Schema abgeleitet werden. Für Klassennamen zur Definition anonymer Instanzen (wie hier Price) gilt dasselbe. Bei Untertypen muss bei benamten Instanzen per `as` <`Klassenname`> (`at` <`Ontologiename`>) die eigentliche Klasse (mit Ontologie) markiert werden. Hier ist dies bei der Instanz greatBritain der Fall, die vom Typ Country ist, einem Untertyp vom erwarteten Location.

In g-dsd wird das Füllen von Attributen durch Pfeile notiert, welche von der Instanz auf den Füllwert zeigen und mit dem Namen des Attributs beschriftet sind. Auch hier muss die Quellontologie nur dann angegeben werden, wenn sie sich nicht eindeutig aus dem Schema ableiten lässt. Abbildung 15 zeigt das Beispiel von oben in g-dsd.

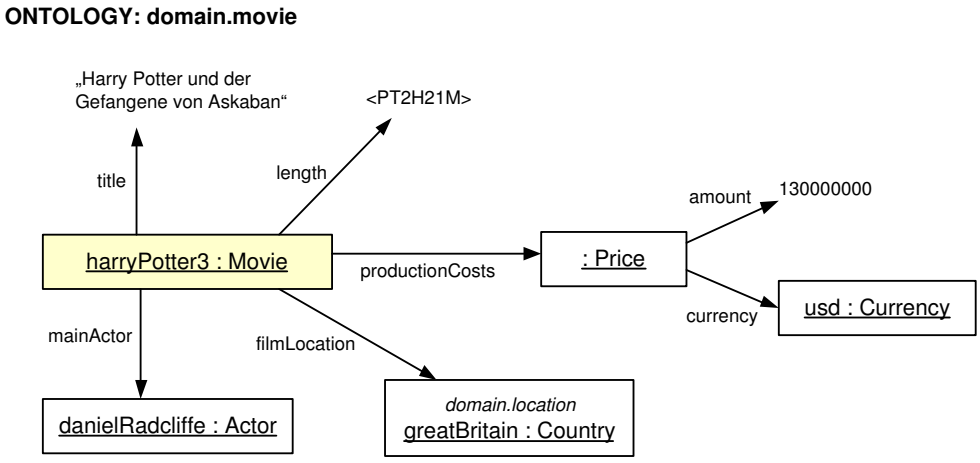


Abbildung 15: Ausfüllen von Attributen in g-dsd.

Generell gilt, dass in g-dsd das Auftauchen einer benamten Instanz nur dann zu einer Neudefinition führt, wenn diese nicht als Füllwert dient (d.h. keine Pfeile auf sie weisen).

In j-dsd werden die Attribute einer Instanz gesetzt, in dem die Attribute des jeweiligen Javaobjekts gesetzt werden. Dabei werden primitive Füllwerte über den entsprechenden Konstruktor des XSD-Types erzeugt, benamte Instanzen durch Aufruf der zugehörigen `create`-Methode referenziert und anonyme Instanzen in einem Vorschritt erzeugt. Die Füllung erfolgt für benamte Instanzen in deren `create`-Methode, für anonyme Instanzen direkt nach deren Erzeugung durch den Konstruktor. Das Beispiel von oben in j-dsd zeigt Abbildung 16.

3.2.4 Füllen listenwertiger Attribute

Beim Füllen listenwertiger Attribute muss eine geordnete Liste von Füllwerten angegeben werden.

In f-dsd geschieht das über eine Reihe von Füllanweisungen der Form <`Listenwertiges Attribut`> `+=` <`Füllwert`> im Instanzattributblock. Die Reihenfolge entspricht dabei der Reihenfolge in der Liste. Dabei gelten dieselben Regeln wie bei der Füllung eines gewöhnlichen Attributs. Im Beispiel:

```
{ontology domain.movie}
```

```
package dsd.instance.domain.movie.Movie;

import dsd.elements.xsd.*;
import dsd.instance.domain.movie.Actor.danielRadcliffe;
import dsd.instance.domain.location.Country.greatBritain;
import dsd.schema.domain.money.Price;
import dsd.schema.domain.movie.Movie;

public class harryPotter3
{
   public static Movie create()
   {
       Movie instance = new Movie();
       instance.instanceName
          = "dsd.instance.domain.movie.Movie.harryPotter3";

       //Anonyme Instanz
       Price price01 = new Price()
       price01.amount = 130000000;
       price01.currency = dsd.instance.domain.money.Currency.usd.create();

       instance.title
           = new XSD_String("Harry Potter und der Gefangene von Askaban");
       instance.length = new XSD_Duration("PT2H21M");
       instance.mainActor = danielRadcliffe.create();
       instance.productionCosts = price01;
       instance.filmLocation = greatBritain.create();

       return instance;
   }
}
```

Abbildung 16: Definition der Instanz `harryPotter3` und Füllen ihrer Attribute in j-dsd.

```
tomHanks as Actor
{
   name = "Tom Hanks",
   awards += oscar,
   awards += goldenGlobe,
   awards += mtvMovieAward
}
```

In g-dsd notiert man bei der Füllung eines listenwertigen Attributs mehrere Pfeile, welche wie in der Schemadefinition mit `{list}` am Pfeilende markiert sind. Die Reihenfolge der Liste ergibt sich aus einer Nummerierung am Pfeilanfang. Abbildung 17 zeigt ein Beispiel.

Da listenwertige Attribut in j-dsd durch eine `ThingList` darstellt werden, müssen beim Füllen des Attributs die Elemente zu dieser Liste hinzugefügt werden. Dies geschieht über die spezielle Methode `addElement` in `ThingList`. Das Beispiel sieht wie folgt aus:

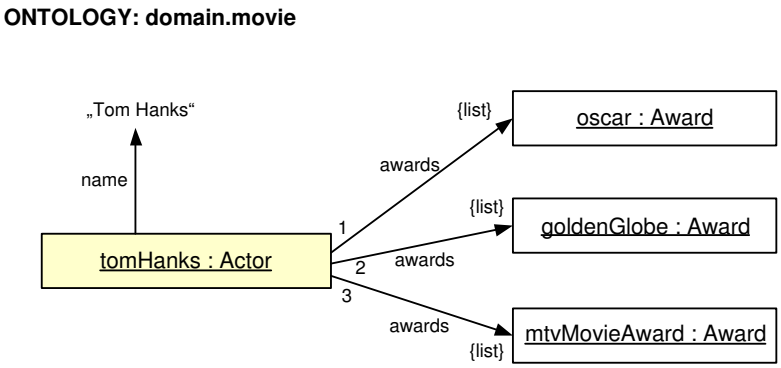


Abbildung 17: Ausfüllen von listenwertigen Attributen in g-dsd.

```
public class tomHanks
{
   public static Actor create()
   {
        Actor instance = new Actor();
        instance.instanceName = "dsd.instance.domain.schema.Actor.tomHanks";
        instance.name = new XSD_String("Tom Hanks");
        instance.awards.addElement(oscar.create());
        instance.awards.addElement(goldenGlobe.create());
        instance.awards.addElement(mtvMovieAward.create());
        return instance;
   }
}
```

3.3 Wertbestimmte und Entitätsklassen

Wie gesehen existiert ein wesentlicher semantischer Unterschied zwischen benamten und anonymen Instanzen. Dieser Unterschied wirkt sich auch auf die zugehörigen Klassen aus: Klassen können entweder nur anonyme Instanzen oder nur benamte Instanzen besitzen. Eine Mischung ist semantisch nicht sinnvoll. Es ist daher wichtig, für jede Klasse bei ihrer Definition anzugeben, von welchem Typ sie ist.

3.3.1 Definition und Eigenschaften

In DSD werden *wertbestimmte* und *Entitätsklassen* unterschieden.

- **Wertbestimmte Klassen.** Klassen, bei denen jede beliebige, gültige Kombination aus Füllwerten zu Instanzen führen, die existierende oder schaffbare Individuen der realen Welt beschreiben, gelten als wertbestimmte Klassen. Ausgehend von einer beliebigen Instanz einer solchen Klasse führt eine (kleine) Änderung an einem der Füllwerte bereits zu einer neuen Instanz, die ein anderes Individuum der Welt beschreibt. Beispiele für solche Klassen sind Price, WeightMeasure oder Printed. Für solche Klassen sind nur anonyme Instanzen sinnvoll. Aufgrund der Orthogonalität der Attribute ist die Angabe von definierenden oder orthogonalen Attributen nicht sinnvoll (alle Attribute können als definierend angesehen werden). Die

Identität der Instanzen ist vollständig durch die Füllwerte bestimmbar. Häufig sind dazu jedoch domänenspezifische Gleichheits- oder Ähnlichkeitsfunktionen nötig (siehe Abschnitt 3.9). Klassen ohne Attribute sind nie wertbestimmt.

- **Entitätsklassen.** Klassen, bei denen nur bestimmte Kombinationen von Füllwerten zu Instanzen führen, die Individuen der Welt beschreiben, gelten als Entitätsklassen. Jede solche Wertekombination beschreibt eine eigenständige *Entität*, die durch einen global eigenständigen Namen gekennzeichnet werden kann. Für solche Klassen sind daher nur benamte Instanzen sinnvoll. Beispiele für solche Klassen sind Movie, Person und Actor. Die Identität der Entitäten ist nicht immer einfach aus den Attributen abzuleiten, denn kleine Änderungen an den Attributen führen nicht zwangsläufig zu neuen Entitäten. Daher sollte der Vergleich den Namen der Entität in Betracht ziehen. Die Verwendung von definierenden Attributen ist dann sinnvoll, wenn die Namen zwar benamt, diese Namen aber nicht zum Teil nicht veröffentlicht sind (siehe Abschnitt 3.4).

3.3.2 Repräsentierung

In f-dsd wird die Markierung einer Entitätsklasse durch Voranstellen des Schlüsselworts `entityclass` vor der Klassendefinition erreicht. Wertbestimmte Klassen erhalten die Markierung `valueclass`. Bei fehlender Markierung wird als Standard eine Markierung mit `entityclass` angenommen.

```
entityclass Person
[
   name: String
]

valueclass Price
[
   amount: Double,
   currency: Currency
]

entityclass Currency []
```

In g-dsd erfolgt die Markierung durch einen tiefgestellten Index am Namen der Klasse: E für Entitätsklassen, V für wertbestimmte Klassen. Auch hier wird als Standard eine Markierung mit E angenommen. Abbildung 18 zeigt ein Beispiel.

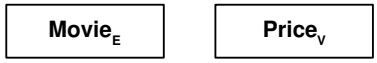


Abbildung 18: Differenzierung zwischen wertbestimmten und Entitätsklassen in g-dsd.

In j-dsd erfolgt die Markierung im statischen Attribut `entityclasses` vom Typ `Vector` in `Thing`. Hier werden alle Namen von Entitätsklassen abgelegt. Die Namen sind dabei vollständig mit Ontologie anzugeben. Nicht abgelegte Klassen gelten als wertbestimmte Klassen. Zum Eintrag in den `Vector` eignet sich ein statischer Initialisierer. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
public class Person extends Thing
{
   static {entityclasses.add("domain.person.Person");}

   public XSD_String name;
}

public class Price extends Thing
{
   public XSD_Double amount;
   public Currency currency;
}

public class Currency extends Thing
{
   static {entityclasses.add("domain.money.Currency");}
}
```

3.4 Öffentliche und teilöffentliche Entitätsklassen

Bisher wurde davon ausgegangen, dass alle benamten Instanzen im öffentlich zugänglichen Instanzenpool abgelegt werden und so jedem zur Verfügung stehen. Dies führt zu einem erheblichen Aufwand in der Community, die sich im Prinzip bei jeder einzelnen Instanz einigen muss. Insbesondere bei Dienstbeschreibungen ist es jedoch sinnvoll, bestimmte Instanzen, die nicht von allgemeinem Interesse sind, zwar (gedanklich) zu benennen, diesen Namen aber nicht im allgemeinen Instanzenpool zu veröffentlichen. VORSICHT: Diese Instanzen sind nach wie vor benamt und daher nicht mit anonymen Instanzen zu verwechseln, denen man keinen Namen geben kann. Ihr Name ist nur nicht veröffentlicht. Man unterscheidet zwei Arten von Entitätsklassen:

- **Öffentliche Entitätsklassen.** In öffentlichen Entitätsklassen ist es nicht erlaubt, in einer Beschreibung, private, unveröffentlichte Instanzen vorauszusetzen. Jede Instanz die verwendet wird, ist von allgemeinem Interesse und muss im Instanzenpool unter einem eindeutigen Namen veröffentlicht werden. Beispiele für solche Klassen sind Movie, Actor und Currency. Da alle verwendeten Instanzen bekannt sind, ist die Verwendung definierender Attribute nicht nötig. Sie können jedoch zur besseren Einhaltung der Bijektionsforderung (siehe Abschnitt 3.8) verwendet werden.
- **Teilöffentliche Entitätsklassen.** In solchen Entitätsklassen ist das Voraussetzen von unveröffentlichten, privaten Instanzen erlaubt. Dies muss beim Vergleich von Beschreibungen berücksichtigt werden. Beispiele für solche Klassen sind Person und Cinema. Definierende Attribute sind in solchen Klassen von besonderer Bedeutung, da häufig nur deklarativ über die Füllwerte dieser Attribute angefragt werden kann.

Als Standard gilt für Entitätsklassen, dass sie teilveröffentlicht sind. Öffentliche Entitätsklassen werden in f-dsd durch voranstellen des Schlüsselworts `public` markiert:

```
public entityclass Actor [...]

entityclass Person [...]
```

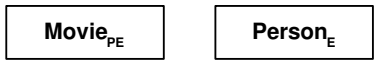


Abbildung 19: Markierung von öffentlichen Entitätsklassen in g-dsd.

In g-dsd erfolgt die Markierung durch Anhängen des Index P an den Klassennamen, wie das Beispiel in Abbildung 19 zeigt.

In j-dsd erfolgt die Markierung im statischen Attribut `publicclasses` vom Typ `Vector` in `Thing`. Hier werden alle Namen von öffentlichen Entitätsklassen abgelegt. Die Namen sind dabei vollständig mit Ontologie anzugeben. Nicht abgelegte Klassen gelten als teilöffentliche Klassen. Zum Eintrag in den `Vector` eignet sich ein statischer Initialisierer. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
public class Actor extends Person
{
   static {entityclasses.add("domain.movie.Actor");}
   static {publicclasses.add("domain.movie.Actor");}
   ...
}

public class Person extends Thing
{
   static {entityclasses.add("domain.person.Person");}
   ...
}
```

3.5 Vererbungsbeschränkungen

Aufgrund der großen semantischen Unterschiede zwischen wertbestimmten und Entitätsklassen und den daraus resultierenden Forderungen an die Instanziierung sowie der Öffentlichkeit von Entitätsklassen, entstehen auch Einschränkungen bei der Vererbung von Klassen:

- Eine Vererbung zwischen einer wertbestimmten Oberklasse und einer Unterklasse als Entitätsklasse oder umgekehrt ist nicht erlaubt.
- Eine teilveröffentlichte Klasse darf nicht von einer öffentlichen Klasse erben.

Generell gilt also, dass die Markierer `entityclass`, `valueclass` und `public` an die Unterklassen „vererbt“ werden. Durch die Überprüfung dieser Eigenschaft können auch prinzipielle Modellierungsfehler beim Erstellen einer Ontologie entdeckt werden. Wichtig: Diese Vererbung ist nur gedacht. In allen Repräsentationsformen müssen die Markierer auch in den Unterklassen vollständig angegeben werden

3.6 Anforderungen an das Füllen von Attributen

Beim Anlegen neuer benamter und anonymer Instanzen gelten bestimmte Anforderungen an das Füllen von Attributen:

- Da wertbestimmte Klassen allein durch die Füllwerte ihrer Attribute bestimmt sind, müssen bei der Definition anonymer Instanzen *alle* Attribute gefüllt werden. Hierdurch kann beispielsweise für OUT-Variablen wertbestimmter Typen erwartet werden, dass die gelieferte Instanz in allen Attributen definiert sein wird.

- Bei der Definition von benamten Instanzen müssen alle definierenden Attribute gefüllt werden, abgeleitete Attribute können gefüllt werden. Orthogonale Attribute sollten bei der Erstdefinition nicht gefüllt werden, sondern können auch noch später außerhalb der Ontologiedefinition zugewiesen werden.

3.7 Gleichheit von Instanzen

Nach der Definition von Instanzen stellt sich die Frage, wann zwei gegebene Instanzen oder Werte gleich sind. Die Gleichheit wird durch das Symbol `==` markiert. Die Bestimmung hängt von der Art ab:

- Die Gleichheit primitiver Werte wird durch eine vorgegebene Gleichheitsfunktion bestimmt. Dabei ist nicht unbedingt eine zeichengenaue Gleichheit nötig, z.B. gilt auch `1.0 == 1.00` oder `<PT1H> == <PT60M>`. In j-dsd ist die Gleichheitsfunktion durch die Methode `equals` definiert. Diese liefert genau dann true, wenn die `compareTo`-Methode 0 liefert (vgl. Abschnitt 2.1.3).

- Zwei benamte Instanzen sind genau dann gleich, wenn sie den gleichen Namen haben. In j-dsd wird das durch die Methode `equals` realisiert, welche den in `instanceName` hinterlegten Instanznamen vergleicht (vgl. Abschnitt 3.2.1).

- Zwei anonyme Instanzen sind genau dann gleich, wenn sie den gleichen Typ besitzen und die Füllwerte der korrespondierenden Attribut gleich sind.

- Eine anonyme Instanz und eine benamte Instanz sind immer ungleich, da sie nach Abschnitt 3.3 von unterschiedlichen Typen sein müssen.

3.8 Forderungen an Instanzen

An die Erstellung von Instanzen werden zwei wichtige Forderungen gestellt, um eine korrekte Verarbeitung insbesondere während des Vergleichs zu gewährleisten. Die erste Forderung, die Bijektionsforderung, ist sehr fundamental, so dass sich hieraus mit Hilfe der oben vorgestellten Gleichheitsregeln drei konkrete Bedingungen ableiten lassen.

Bijektionsforderung
Zwei Instanzen müssen genau dann gleich sein, wenn sie dasselbe Individuum der realen Welt beschreiben.

Dies ist die wichtigste Forderung. Sie verlangt eine eineindeutige Abbildung zwischen den Individuen der realen Welt und den Instanzen (benamt und anonym) in den Ontologien. Sie ist von großer Bedeutung für den Vergleichsprozess, da dieser darauf angewiesen ist, dass einerseits die Bedeutung der Instanzen eindeutig definiert ist, andererseits Individuen der realen Welt eindeutig benannt sind. Aus der Bijektionsforderung lässt sich folgende Bedingung ableiten:

- *Bedingung nach eindeutigen Namen* (engl. auch *Unique Name Assumption*). Da die Gleichheit auf benamten Instanzen allein durch den Namen bestimmt ist, lässt sich aus der Bijektionsforderung direkt ableiten, dass kein Individuum der realen Welt durch zwei oder mehr benamte Instanzen beschrieben sein darf. Daher ist bei der Veröffentlichung einer neuen benamten Instanz darauf zu achten, dass nicht schon eine benamte Instanz für dieses Individuum existiert.

Für Klassen mit definierenden Attributen gilt eine weitere Forderung, welche deren festlegenden Charakter in der Art eines Schlüssels bestimmt.

Schlüsselforderung
Bei Klassen mit definierenden Attributen sind Instanzen eineindeutig durch die Kombination der Füllwerte in den definierenden Attributen bestimmt.

Auch hieraus lassen sich drei Bedingungen ableiten:

- *Eindeutigkeit.* Zwei benamte Instanzen mit unterschiedlichem Namen von Klassen mit definierenden Attributen dürfen sich nicht in allen Füllwerten ihrer definierenden Attribute gleichen.
- *Ableitbarkeit.* Zwei Instanzen von Klassen mit definierenden Attributen und gleichen Füllwerten in allen diesen Attributen dürfen sich in keinem Füllwert eines ableitbaren Attributs unterscheiden.

Diese Forderung stellt sicher, dass in Klassen mit definierenden Attributen diese als Schlüssel fungieren

3.9 Domänenspezifische Gleichheit, Ähnlichkeit und Ordnung auf Instanzen

Für Klassen besteht die Möglichkeit, eigene Relationen zu definieren. Hierdurch wird auf eine domänenspezifische Weise festgelegt, wann zwei Instanzen gleich sind, wie ähnlich sie sind oder in welcher Ordnung sie stehen. Analog zur Definition von Klassen und benamter Instanzen muss sich die Community auch bei der Definition solcher Relationen einigen. Dennoch ist die Verwendung dieser Relationen besonders in Anfragebeschreibungen nicht zwingend. Hier können über das Konzept der deklarativen Menge jederzeit persönliche Vergleichsfunktionen definiert werden. Domänenspezifische Relationen sind insbesondere für wertbestimmte Klassen von großer Bedeutung.

Die Definition von domänenspezifischen Relationen kann nur in j-dsd erfolgen. Dazu kann in der entsprechenden Klasse eine oder mehrere der folgenden Methode angelegt werden.

- `boolean domEquals(Thing t).` Überprüft, ob `this` und `t` semantisch gleich sind. Wenn ja, wird `true` zurückgeliefert, falls nein `false`.
- `double domSimilar(Thing t).` Überprüft, in welchem Maße `this` und `t` semantisch ähnlich sind. Der Grad der Ähnlichkeit aus dem Intervall $[0, 1]$ wird zurückgeliefert.
- `int domCompareTo(Thing t).` Überprüft, ob `this` semantisch kleiner, gleich oder größer als `t` ist. Entsprechend liefert die Methode -1, 0 oder +1.
- `double domFCompareTo(Thing t).` Überprüft, in welchem Maße `this` semantisch kleiner, gleich oder größer als `t` ist. Entsprechend wird ein Wert aus [-1,1] zurückgeliefert.

Falls domänenspezifische Relationen definiert sind, können diese auch in f-dsd und g-dsd verwendet werden. Hierzu stehen bei der Definition von Bedingungen zusätzliche Operatoren `d==, d~==, d<=, d~<= etc. zur Verfügung` (siehe auch Abschnitt 4.2.1).[7]

[7] Die Verwendung domänenspezifischer Funktionen ist zur Zeit noch nicht möglich.

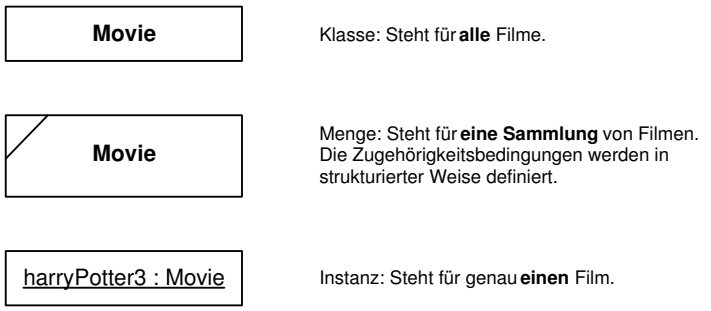


Abbildung 20: Mengen als Zwischending zwischen Klassen und Instanzen.

4 Definieren von Mengen

Mengen von Instanzen stellen ein fundamentales Konzept von DSD dar. Sie werden immer dann eingesetzt, wenn es nicht möglich oder sinnvoll ist, eine einzelne Instanz mittels ihres Namens bzw. ihrer Füllwerte anzugeben. Im Rahmen von Dienstbeschreibungen ist dies allgegenwärtig und tritt an folgenden Stellen auf:

- *In Angebotsbeschreibungen.* Oft kann ein Dienst nicht nur einen einzigen Effekt erzielen, sondern ist prinzipiell in der Lage, eine ganze Reihe von Effekten zu erreichen, die durch ein oder mehrere Parameter eingestellt werden können. Aus diesem Grund kann die Dienstbeschreibung nicht ausschließlich als Instanz angegeben werden, sondern enthält an bestimmten Stellen auch Mengen von Instanzen, aus denen vor und während der Dienstnutzung Elemente durch Füllen von Variablen ausgewählt werden.
- *In Anfragebeschreibungen.* Wie schon in Abschnitt 1.1.4 eingeführt, will der Dienstnehmer in der Regel eine bestimmte Funktionalität erbracht wissen und denkt dabei nicht zwangsläufig an einen bestimmten, einzelnen Dienst. Typischerweise sind auch mehrere unterschiedliche Dienste zur Erbringung dieser Funktionalität geeignet. Es ist daher für ihn wichtig, eine Menge geeigneter Dienste aufzustellen. Die Präferenzen für die einzelnen Dienste können dann durch eine unscharfe Zugehörigkeitsfunktion ausgedrückt werden. Durch diesen Ansatz erreicht man eine vollständige Integration der Präferenzen in die Anfragebeschreibung.

4.1 Deklarative Mengen

Mengen stellen ein Zwischending zwischen Klassen und Einzelinstanzen dar: Klassen stehen stellvertretend für *alle* Individuen eines Typs, Instanzen stehen für *genau ein* Individuum. Instanzmengen hingegen enthalten eine beliebige Sammlung von Instanzen und können daher keine, eine, mehrere oder alle Instanzen eines Typs enthalten. Abbildung 20 zeigt diesen Zusammenhang am Beispiel der Klasse Movie graphisch. In g-dsd werden Mengen wie Klassen notiert, die einen kleinen Querstrich in der linken oberen Ecke besitzen.

Die Bestimmung, welche Instanzen zu einer Menge gehören, erfolgt in DSD deklarativ, d.h. durch Angabe einer Funktion, welche für jede Instanz die Zugehörigkeit zur Menge liefert. Lässt man für eine solche *Zugehörigkeitsfunktion* beliebige Funktionen zu, so erhält man einerseits zwar die größtmögliche Ausdruckskraft, andererseits führt dies zu enormen Problemen: Erstens ist das Aufstellen solcher Funktionen für menschliche Benutzer sehr schwierig, da keinerlei Vorgaben existieren, zweitens kann der Vergleicher mit solchen beliebigen Funktionen nur schwer umgehen, insbesondere

wenn Funktionen in der Anfrage auf Funktionen im Angebot treffen. Solche Situationen führen leicht zu unentscheidbaren Problemen.

In DSD wurde daher bewusst eine Strukturierung (und damit auch Einschränkung) der möglichen Zugehörigkeitsfunktionen vorgenommen, indem bestimmte Beschreibungselemente zur Definition von Mengen vorgegeben wurden. Man unterscheidet Bedingungen und Strategien. Diese sollen im Folgenden vorgestellt werden.

4.1.1 Typbedingung – Type Condition

Die Typbedingung bestimmt den Typ, den die Elemente der Menge haben müssen, d.h. eine Instanz kann nur dann Element der Menge sein, wenn sie von der in der Bedingung angegeben Klasse abgeleitet ist. Die Angabe einer Typbedingung ist für eine Menge verpflichtend. Man sagt dann auch, die Menge habe den angegebenen Typ.

In g-dsd werden Mengen generell wie Klassen notiert und mit einem kleinen Querstrich in der linken, oberen Ecke gekennzeichnet. Die Typbedingung findet sich als Name der Klasse im Kästchen. Stammt die Klasse aus einer fremden Ontologie, wird deren Name wie gewohnt in kleiner, kursiver Schrift über dem Klassennamen notiert. Abbildung 21 zeigt drei Beispiele.

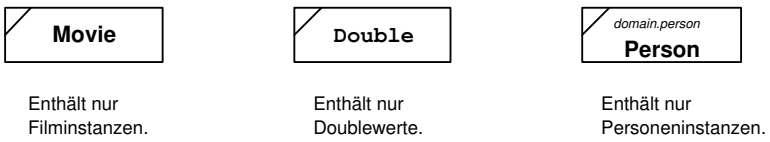


Abbildung 21: Notation von Mengen und Typbedingung in g-dsd.

In f-dsd werden Mengen durch `set of <Typbedingung>` notiert, wobei die Typbedingung den Namen der Klasse darstellt. Stammt die Klasse aus einer fremden Ontologie, kann dies durch ein anschließendes `at <Ontologiename>` gekennzeichnet werden. Mengen werden immer in runden Klammern notiert.

```
(set of Movie)
(set of Double)
(set of Person at domain.person)
```

In j-dsd wird ein Trick angewendet, um Mengen abbilden zu können. Dies ist nötig, da später Mengen an die Position von Einzelinstanzen eingesetzt werden können müssen (siehe dazu Kapitel 6.3). Daher erbt die allgemeine Oberklasse `Thing` von der Javaklasse `Set` aus `dsd.elements`. In `Set` gibt es ein Java-Attribut `boolean isSet`, welches festlegt, ob ein Javaobjekt eine DSD-Instanz oder eine DSD-Menge repräsentiert. DSD-Mengen mit der Typbedingung k werden daher als Javaobjekte der Klasse k dargestellt, wobei `isSet` auf `true` gesetzt wird. Die Beispiele von oben sehen in j-dsd daher wie folgt aus:

```
Movie movieSet = new Movie();
movieSet.isSet = true;

XSD_Double doubleSet = new XSD_Double();
doubleSet.isSet = true;
```

```
import dsd.schema.domain.person.Person;
Person personSet = new Person();
personSet.isSet = true;
```

4.1.2 Direkte Bedingung – Direct Condition

Direkte Bedingungen einer Menge bezeichnen Bedingungen, die direkt an die potenziellen Elemente (d.h. nicht an deren Attribute) gerichtet sind. Hierzu stehen alle Vergleichsoperatoren wie `==`, `<=` etc. zur Verfügung. Zudem kann mit dem Operator `in` auf Enthaltensein in einer aufgelisteten Menge verglichen werden. Die Angabe direkter Bedingungen ist optional. Prinzipiell können auch domänenspezifische Operation (beginnen mit `d`) verwendet werden (siehe Abschnitte 3.9).[8] Es können null, eine oder mehrere direkte Bedingungen angegeben werden. Eine Instanz kann nur dann zur Menge gehören, wenn es alle diese Bedingung erfüllt, d.h. die Bedingungen werden konjunktiv verknüpft.

In g-dsd werden direkte Bedingungen einer Menge in das Kästchen aufgenommen und durch eine Linie abgetrennt. Die Syntax ist dabei `<Operator> <Vergleichsinstanz/-wert>`. Abbildung 22 zeigt drei Beispiele.

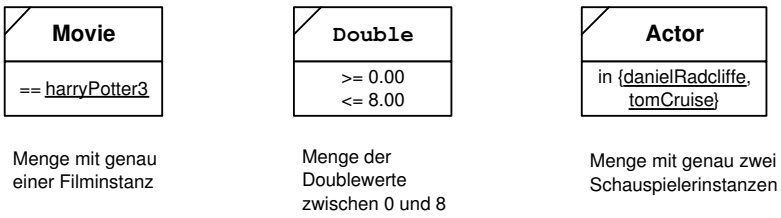


Abbildung 22: Beispiele für Mengen mit direkten Bedingungen.

In f-dsd werden direkte Bedingungen durch `with elements <Operator> <Vergleichsinstanz/-wert>` notiert:

```
(set of Movie
 with elements == harryPotter3)

(set of Location
 with element d~== karlsruhe)

(set of Double
 with elements >= 0.00
 with elements <= 8.00)

(set of Actor
 with elements in {danielRadcliffe, tomCruise})
```

j-dsd verwendet ein spezielles Javaattribut `Vector directConditions` in `Set` zur Aufnahme der direkten Bedingungen. Für jede Bedingung muss hier ein Javaobjekt vom Typ `DirectCondition` eingefügt werden. `DirectCondition` befindet sich ebenfalls in `dsd.elements` und hat zwei Attribute: `String operator` und `Object value`. Als `operator` kommen die oben erwähnten Operatoren in Frage, `value` ist entweder ein `Thing` oder ein `Vector` mit `Thing`s, je nachdem ob der Operator `in`

[8] Domänenspezifische Operationen werden zur Zeit noch nicht unterstützt.

verwendet wurde. `DirectCondition` verfügt auch über eine Methode `test`, welche testet, ob eine Instanz bzw. ein Wert die Bedingung erfüllt. Zwei der Beispiele von oben sehen damit in j-dsd wie folgt aus:

```
Movie movieSet = new Movie();
movieSet.isSet = true;
movieSet.directConditions = new Vector();
DirectCondition movie_dc01 = new DirectCondition();
movie_dc01.operator = "==";
movie_dc01.value = dsd.instance.domain.movie.Movie.harryPotter3;
movieSet.directConditions.add(movie_dc01);

Actor actorSet = new Actor();
actorSet.isSet = true;
actorSet.directConditions = new Vector();
DirectCondition actor_dc01 = new DirectCondition();
actor_dc01.operator = "in";
actor_dc01.value = new Vector();
actor_dc01.value.add(dsd.instance.domain.movie.Actor.danielRadcliffe);
actor_dc01.value.add(dsd.instance.domain.movie.Actor.tomCruise);
actorSet.direcConditions.add(actor_dc01);
```

4.1.3 Attributbedingung – Property Condition

Für Mengen von nicht-primitiven Typen stehen neben direkten Bedingungen auch Attributbedingungen zur Verfügung. Diese überprüfen potenzielle Elemente anhand ihrer ausgefüllten Attribute. Attributbedingungen bestehen immer aus einem Attributnamen, welcher aus der Klasse der Menge stammt, sowie einer *Zielmenge*. Es gilt: Eine Instanz kann nur dann Element der Menge sein, wenn das Attribut aus der Bedingung bei ihr gefüllt ist und der Füllwert Element der Zielmenge ist. Hierdurch können verschachtelte, deklarative Mengen aufgebaut werden. Attributbedingungen sind optional. Treten mehrere auf, werden diese standardmäßig konjunktiv verknüpft (siehe dazu auch Abschnitt 4.1.5).

In g-dsd werden Attributbedingungen durch einen Pfeil notiert, der den Namen des Attributs trägt und auf die Zielmenge zeigt. Abbildung 23 zeigt ein Beispiel.

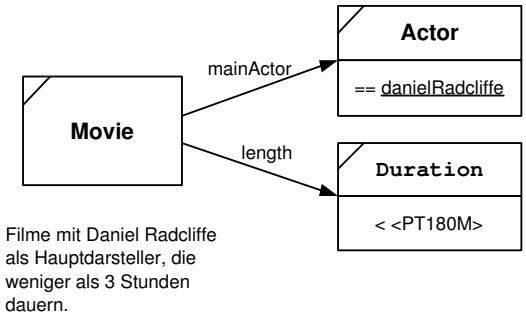


Abbildung 23: Beispiele für eine Menge mit Attributbedingungen.

In f-dsd steht für Attributbedingungen das Konstrukt `where <Attributname> in <Zielmenge>` zur Verfügung. Das Beispiel sieht also wie folgt aus:

```
(
  set of Movie
  where mainActor in
  (
    set of Actor
    with elements == danielRadcliffe
  )
  where length in
  (
    set of Duration
    with elements < <PT180M>
  )
)
```

Als Abkürzung können einelementige Zielmengen durch das Konstrukt `where <Attributname> == <einziges Element der Zielmenge>` beschrieben werden. Das Beispiel kann also wie folgt verkürzt werden:

```
(
  set of Movie
  where mainActor == danielRadcliffe
  where length in
  (
     set of Duration
     with elements < <PT180M>
  )
)
```

In j-dsd werden die gewöhnlichen Attribute der Klasse genutzt, um Attributbedingungen auszudrücken. Es gilt: Attribute einer Menge (d.h. einer Instanz mit `isSet == true`), die nicht `null` sind, gelten als Attributbedingungen. Das Beispiel von oben sieht also in j-dsd wie folgt aus, wenn man annimmt, dass die Actor- und `Duration`-Menge zuvor bereits in `actorSet` und `durationSet` definiert sind:

```
Movie movieSet = new Movie();
movieSet.isSet = true;
movieSet.mainActor = actorSet;
movieSet.length = durationSet;
```

Achtung: Eine Abkürzung wie in f-dsd existiert in j-dsd nicht. Einelementige Zielmengen müssen stets über direkte Bedingungen ausgeschrieben werden.

4.1.4 Fehlstrategie – Missing Strategy

Generell gilt, dass eine Instanz nur dann eine Attributbedingung erfüllen kann, wenn bei ihr das geforderte Attribut auch gefüllt ist. Eine alternative Fehlstrategie modifiziert dieses Verhalten. Es gibt drei Arten von Fehlstrategien:

- `assume_failed.` Bezeichnet den Standardfall. Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung als fehlgeschlagen angesehen, d.h. der Wahrheitswert `false` wird angenommen.
- `assume_fulfilled.` Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung als erfüllt angesehen, d.h. der Wahrheitswert `true` wird angenommen.
- `ignore.` Wenn der entsprechende Füllwert fehlt, wird die Attributbedingung ignoriert, d.h. der zusätzliche Wahrheitswert `neutral` wird angenommen.[9]

Unterschiede zwischen den Rückgabewerten `neutral` und `true` treten nur bei geänderter Verbindungsstrategie auf (siehe Abschnitt 4.1.5).

Wichtig: Nach Abschnitt 3.6 müssen bestimmte Attribute stets gefüllt werden. Die Angabe von Fehlstrategien für Bedingungen an solche Attribute ist daher nicht sinnvoll und sollte vermieden werden. Daher sind Fehlstrategien nur für abgeleitete oder orthogonale Attribute von Entitätsklassen sinnvoll.

In g-dsd wird die Fehlstrategie durch eine Markierung am Pfeil der Attributbedingung kenntlich gemacht. Hierbei steht ein Minuszeichen (oder keine Markierung) für den Standardfall `assume_failed`, eine Pluszeichen für `assume_fulfilled` und ein ausgefüllter Kreis für `ignore`. Abbildung 24 zeigt ein Beispiel für die `ignore`-Strategie.

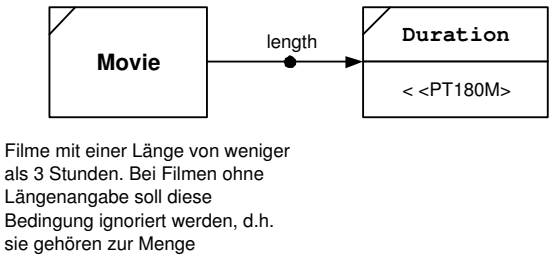


Abbildung 24: Beispiel für die Fehlstrategie `ignore`.

In f-dsd wird die Fehlstrategie direkt nach dem `where` der Attributbedingung in geschweifte Klammern gesetzt. Die Syntax lautet `{when missing <Fehlstrategiename>}`, wobei im Namen der Strategie Unterstriche durch Leerzeichen ersetzt werden. Das Beispiel von oben sieht also in f-dsd wie folgt aus:

```
(
  set of Movie
  where {when missing ignore} length in
  (
    set of Duration
    with elements < <PT180M>
  )
)
```

In j-dsd steht zur Markierung der Fehlstrategie eine nicht-statische Hashtabelle `missingStrategy` in `Set` zur Verfügung. Diese wird mit Tupeln der Form `(<Attributname>, <Fehlstrategiename>)`

[9] `neutral` ist das neutrale Element für alle booleschen Operationen. Es gilt dabei: $x \wedge$ `neutral` $= x$, $x \vee$ `neutral` $= x$, $\neg$`neutral` $=$ `neutral` mit $x \in \{$`true`, `false`, `neutral`$\}$.

gefüllt. Fehlende Attribute haben implizit die Standardstrategie `assume_failed`. Ist `missingStrategy` `null`, so haben alle Attribute diese Standardstrategie. Das Beispiel von oben sieht in j-dsd also wie folgt aus (sofern `durationSet` bereits definiert ist):

```
Movie movieSet = new Movie();
movieSet.isSet = true;
movieSet.missingStrategy = new Hashtable();
movieSet.missingStrategy.put("length", "ignore");
movieSet.length = durationSet;
```

4.1.5 Verbindungsstrategie – Connecting Strategy

Die Verbindungsstrategie modifiziert das Verhalten, wie die einzelnen Attributbedingungen verknüpft werden. Standardmäßig werden diese konjunktiv verbunden. Die Strategie stellt einen booleschen Ausdruck bestehend aus den als Attributbedingung auftretenden Attributnamen und den Operatoren `and` und `or` da. Es gilt: Eine Instanz kann nur Element einer Menge sein, wenn der Ausdruck der Verbindungsstrategie nicht zu `false` evaluiert, wobei die Attributnamen durch die Einzelergebnisse der entsprechenden Attributbedingungen (also `true`, `false` oder `neutral`) ersetzt werden.

In g-dsd wird eine alternative Verbindungsstrategie in kursiver Schrift in das Kästchen der Menge geschrieben und durch eine horizontale Linie abgetrennt. Abbildung 25 zeigt ein Beispiel.

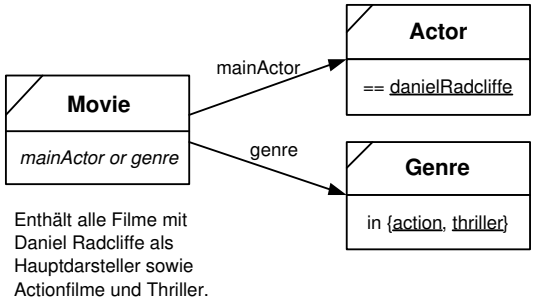


Abbildung 25: Beispiel für eine alternative Verbindungsstrategie.

In f-dsd wird eine Verbindungsstrategie am Ende der Mengendefinition durch `combine by <Verbindungsausdruck>` notiert. Dieser muss vollständig geklammert sein, d.h. je zwei Operanden werden in runden Klammern umschlossen. Das Beispiel von oben sieht also wie folgt aus:

```
(
  set of Movie
  where mainActor == danielRadcliffe
  where genre in
  (
    set of Genre
    with elements in {action, thriller}
  )
  combine by (mainActor or genre)
)
```

Die Notation der Verbindungsstrategie in j-dsd wird in Abschnitt 4.2.3 vorgestellt.

4.1.6 Typvergleichsstrategie – Type Check Strategy

Standardmäßig müssen die Elemente einer Menge vom Typ der Menge oder einem Untertyp von diesem sein. Durch Angabe einer alternativen Typvergleichsstrategie kann dies so abgeschwächt werden, dass auch Oberklassen erlaubt sind. Folgende Strategien stehen zur Verfügung:

- `=`. Bezeichnet den Standard, bei dem nur eine exakte Typübereinstimmung bzw. eine Unterklassenbeziehungen erlaubt ist.
- `super`. Elemente der Menge dürfen auch von einer Oberklassen der Mengenklasse instanziiert sein.
- `super[`n`,1]`. Elemente der Menge dürfen auch von einer Oberklassen der Mengenklasse instanziiert sein, aber maximal über n Vererbungsbeziehungen.

In g-dsd wird eine optionale Typvergleichsstrategie direkt im rechten unteren Bereich des Kästchens notiert. Die Strategie = kann dabei als Standard weggelassen werden. Abbildung 26 zeigt ein Beispiel.

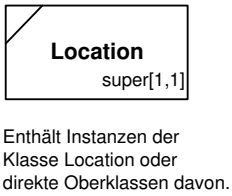


Abbildung 26: Beispiel für eine alternative Typvergleichsstrategie.

In f-dsd wird die Typvergleichsstrategie durch `or supertype` nach dem Klassennamen der Menge notiert. Eine evtl. maximale Länge von Vererbungsbeziehungen folgt in eckigen Klammern. Das Beispiel sieht also wie folgt aus:

```
(set of Movie or supertype[1,1])
```

In j-dsd kann die Typvergleichsstrategie durch das Attribut `String typeCheckStrategy` in `Set` eingestellt werden. Werte sind die oben vorgestellten Kurzbezeichnungen. Im Beispiel:

```
Movie movieSet = new Movie();
movieSet.isSet = true;
movieSet.typeCheckStrategy = "super[1,1]";
```

Zu beachten ist der Zusammenhang zwischen einer geänderten Typvergleichsstrategie und Attributbedingungen. Es gilt folgendes: Sei M eine Menge vom Typ T mit einer Typvergleichsstrategie, die Obertypen zulässt. Sei S ein Obertyp von T. Gibt es in M Attributbedingungen bezüglich Attribute, die nicht in S vorkommen, so werden diese für Instanzen mit dem speziellsten Typ S nicht überprüft. Wichtig: Für diese kommt nicht die Fehlstrategie zum Einsatz. Sei weiterhin R ein Untertyp von S und damit eine Schwesterklasse von T. Ist in R und T ein gleichlautendes Attribut a definiert, das nicht in S vorkommt, so wird dieses nicht herangezogen, falls in M hieran eine Attributbedingung geknüpft ist, die auf einer Instanz aus R nicht überprüft werden kann.

4.1.7 Test auf Mengenzugehörigkeit

Generell gilt, dass genau die Instanzen bzw. Werte zu einer Menge gehören, die alle Bedingungen unter Berücksichtigung der Strategien erfüllen. Dabei wird das Ergebnis der Typbedingung, das aus den direkten Bedingungen und das aus den Attributbedingungen immer konjunktiv verknüpft. Am Ende steht daher auf jeden Fall einer der booleschen Werte `true` oder `false`. `neutral` kann nicht mehr vorkommen, da zumindest die Typbedingung nicht `neutral` liefert.

In j-dsd existiert in `Set` eine Methode zur Bestimmung der Mengenzugehörigkeit: `double contains (Thing t)`. Diese testet, ob eine gegebene Instanz `t` in der Menge liegt. Wenn ja, liefert sie 1.0, wenn nicht 0.0. Wichtig ist dabei, dass `t` keine Mengen oder Variablen enthalten darf (siehe Kapitel 6.3). `contains` stützt sich auf drei private Hilfsmethoden `checkType`, `checkDirectConditions` und die rekursive `checkPropertyConditions`, um das Ergebnis zu bestimmen.

4.2 Unscharfe deklarative Mengen

Bei den bisher vorgestellten Mengen handelt es sich um *scharfe* Mengen, d.h. für jede Instanz kann angegeben werden, ob sie sich vollkommen in der Menge befindet oder nicht. Zum Aufstellen von Anfragebeschreibungen ist dies nicht ausreichend. Hier will der Dienstnehmer über den Grad der Mengenzugehörigkeit seine Präferenzen unter den Elementen der Menge ausdrücken können. Aus diesem Grund wurde DSD um das Konzept *unscharfer Mengen* erweitert. Unscharfe Mengen lassen sich durch eine kontinuierliche Zugehörigkeitsfunktion definieren, die jeder Instanz einen Wert aus dem Intervall $[0.0, 1.0]$ zuweist. Dabei bedeutet 0, dass die Instanz nicht zur Menge gehört, wohingegen ein Wert > 0 die graduelle Zugehörigkeit angibt. Scharfe Mengen sind ein Spezialfall unscharfer Mengen, die nur die Zugehörigkeiten 0 und 1 verwenden. Beide Vorgehensweisen können daher gemischt verwendet werden.

Um unscharfe Mengen definieren zu können, wurden zusätzliche Beschreibungselemente im Bereich der direkten Bedingungen sowie der drei Strategietypen eingeführt. Diese werden im Folgenden vorgestellt.

4.2.1 Unscharfe direkte Bedingungen

Unscharfe direkte Bedingungen für Mengen primitiver Typen werden möglich, indem die in Abschnitt 2.1.4 eingeführten unscharfen Vergleichsoperatoren ∼`==`, ∼`<=` etc. als Operatoren zugelassen werden. Auch eine selbstdefinierte Abweichung durch eine Wertangabe in eckigen Klammern ist erlaubt. Die Notation in den drei Repräsentationsformen ändert sich dabei nicht. Beispiele in f-dsd sehen wie folgt aus:

```
(
  set of DateTime
  with elements ~== <2004-08-24T14:00>
)

(
  set of Double
  with elements ~>=[6.00] 7.00
  with elements ~<=[9.00] 8.00
)
```

Für Mengen von komplexen Typen existieren zwei Möglichkeiten unscharfe Bedingungen zu formulieren: direkte Angabe von Zugehörigkeitswerten in aufgelisteten Mengen beim `in`-Operator und domänenspezifische Ähnlichkeitsfunktionen (siehe Abschnitt 3.9). Im ersten Fall wird jedem Element der Vergleichsmenge konkret ein Wert aus $[0, 1]$ zugewiesen, der in eckigen Klammern hinter dem Element notiert wird. In f-dsd sieht das beispielsweise wie folgt aus:

```
(
  set of Genre
  with elements in {thriller[1.0], action[0.8], comedy[0.2]}
)

(
  set of Date
  with elements in {<2004-01-01>[0.75], <2005-01-01>[0.5]}
)
```

Im zweiten Fall wird eine domänenspezifische Ähnlichkeitsfunktion `domSimilar` eingesetzt, die speziell für diese Klasse entwickelt wurde. Eine solche Funktion muss direkt in j-dsd als Methode implementiert werden. Eine beispielhafte Nutzung einer solchen Methode sieht dann in f-dsd wie folgt aus:

```
(
  set of Movie
  with elements domSimilar(harryPotter3)
)
```

In dieser Menge sind beispielsweise Filme, die ähnlich zum angegebenen Film sind. Die Ähnlichkeit wurde dabei von einem Domänenexperten festgelegt und könnte etwa die Darsteller, den Regisseur und das Genre des Films in Betracht ziehen.

4.2.2 Unscharfe Fehlstrategien

Für unscharfe Mengen wurden die Fehlstrategien um eine weitere, generische Strategie ergänzt: `assume_value[`n`]`. Im Falle eines fehlenden Füllwerts wird der in Klammern angegebene Wert n als Erfüllungsmaß herangezogen. n stammt dabei aus dem Intervall $[0, 1]$, wobei $n = 0$ der Strategie `assume_failed`, $n = 1$ der Strategie `assume_fulfilled` entspricht.

In g-dsd erfolgt die Notation durch ein Pluszeichen mit dem daneben gestellten Wert n in eckigen Klammern. Ein Beispiel zeigt Abbildung 27.

In f-dsd notiert man diese Strategie wie gewohnt. Im Beispiel:

```
(
  set of Movie
  where {when missing assume value[0.8]} duration in
  (
    set of Duration
    with elements ~== <PT90M>
  )
)
```

In j-dsd erfolgt der Eintrag wie gehabt in der Hashtable `missingStrategy`.

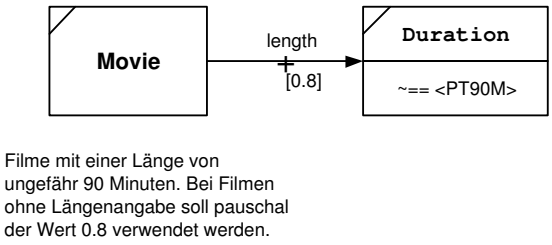


Abbildung 27: Unscharfe Fehlstrategie `assume_value` in g-dsd.

4.2.3 Unscharfe Verbindungsstrategien

Da die Einzelergebnisse der Attributbedingungen in unscharfen Mengen keine booleschen Werte, sondern Zahlen aus dem Intervall $[0, 1]$ sind, muss auch die Vorschrift zu ihrer Verrechnung angepasst werden. `and`-Verknüpfungen werden durch die Multiplikation „$\cdot$“ ersetzt. Für den Spezialfall scharfer Werte, d.h. 0 für `false` und 1 für `true` liefert sie dasselbe Ergebnis. `or`-Verknüpfungen werden durch eine modifizierte Addition „$\oplus$“ ersetzt. Sie ist definiert als $a \oplus b = a + b - ab$ für $a, b \in [0, 1]$. Auch sie liefert für den Spezialfall scharfer Werte das korrekte Ergebnis. Insgesamt stehen folgende unscharfe Operatoren für Werte $x_1, x_2, \ldots \in [0, 1]$ zur Verfügung:[10]

- *Konjunktion*: `(`x_1 `and` x_2`)` bzw. `(`x_1 `mul` x_2`)`. Steht für die gewöhnliche Multiplikation, d.h. das Ergebnis berechnet sich zu $x_1 \cdot x_2$.
- *Disjunktion*: `(`x_1 `or` x_2`)` bzw. `(`x_1 `add` x_2`)`. Steht für die modifizierte Addition, d.h. das Ergebnis berechnet sich zu $x_1 \oplus x_2$.
- *Extremale Konjunktion*: `min(`x_1`, ...,` x_n`)`. Steht für die Minimalwertbildung, die für scharfe Werte dasselbe Ergebnis liefert wie die Konjunktion.
- *Extremale Disjunktion*: `max(`x_1`, ...,` x_n`)`. Steht für die Maximalwertbildung, die für scharfe Werte dasselbe Ergebnis liefert wie die Disjunktion.
- *Gewichtete Summe*: `(`λ_1 `*` x_1 `+ ... +` λ_n `*` x_n`)` wobei $\lambda_1 + \ldots + \lambda_n = 1$. Gewichtet die Einzelergebnisse unterschiedlich und addiert die Werte. Die gewichtete Summe ist dadurch als disjunktive Verrechnung zu werten. Das Ergebnis berechnet sich zu: $\lambda_1 \cdot x_1 + \ldots + \lambda_n \cdot x_n$.
- *Exponentielle Verstärkung*: `exp(`x_1`,` λ`)`. Verstärkt die „Wichtigkeit“ des Wertes x_1 oder schwächt sie ab. $\lambda = 1$ bewirkt keine Änderung, $\lambda < 1$ eine Abschwächung, $\lambda > 1$ eine Verstärkung. Die Berechnung erfolgt zu $(x_1)^\lambda$.

Ist einer der Operanden x_i durch die Fehlstrategie `ignore neutral`, so wird er als neutrales Element der jeweiligen Operation aufgefasst. Bei `mul`, `add`, `min`, `max` bzw. der gewichteten Summe werden alle auftretenden `neutral`-Operanden ignoriert, außer alle Operanden sind `neutral`; in dem Fall liefert die Operation selbst `neutral`. Bei der gewichteten Summe werden die Gewichte der `neutral`-Operanden außerdem gleichmäßig auf die anderen Operanden verteilt. `exp` liefert immer `neutral`, falls der Operand `neutral` ist.

Aus diesen Operationen kann ein beliebiger Ausdruck aufgebaut und als Verbindungsstrategie eingesetzt werden. Die Notation erfolgt in f-dsd und g-dsd unverändert. Ein Beispiel in f-dsd könnte (abgekürzt) wie folgt aussehen:

[10] Weitere unscharfe Verbindungsstrategien sind in Planung.

```
(
  set of Movie
  where mainActor in ...
  where director in ...
  where genre in ...
  combine by (0.8 * (exp(mainActor,2) and director) + 0.2 * genre)
)
```

In j-dsd wird eine alternative Verbindungsstrategie im Attribut `connectingStrategy` von `Set` abgelegt. Als Typ dient die abstrakte Javaklasse `Operation` aus dem Paket `dsd.elements.operation`. Von ihr erben alle konkreten Operationen: `Mul`, `Add`, `Min`, `Max`, `Exp` und `WS` (für weighted sum). Jede dieser Operationen hat eine gewisse Anzahl von Operanden, die selbst wieder von Typ Operation sind, sowie evtl. zusätzliche Parameter. Hierdurch wird es möglich, einen rekursiven Operatorbaum aufzubauen. Ein besondere Operation stellt `PropertyConditionValue` dar: sie repräsentiert ein Blatt im Operatorbaum als Name einer einzelnen Attributbedingung. Zur Auswertung einer Operation existiert in `Operation` die abstrakte Methode `evaluate`, die von den konkreten Operation implementiert wird. Innere Knoten verrechnen dabei ihre Operanden entsprechend des Operators, während Blätter das Ergebnis der entsprechenden Attributbedingung zurückliefern. Hierzu benötigt `evaluate` eine Hashtable mit diesen Werten als Parameter. In j-dsd sieht die Verbindungsstrategie `((0.3 * mainActor + 0.7 * genre) and director)` sieht in j-dsd also wie folgt aus. Beachte, dass das scharfe `and` durch `mul` sowie das scharfe `or` durch `add` ersetzt werden:

```
Movie movieSet = ...
PropertyConditionValue pcv_1 = new PropertyConditionValue();
pcv_1.pcName = "mainActor";
PropertyConditionValue pcv_2 = new PropertyConditionValue();
pcv_2.pcName = "genre";
WS ws_3 = new WS();
ws_3.ops = new Vector();
ws_3.lambdas = new Vector();
ws_3.ops.add(pcv_1);
ws_3.lambdas.add(new Double(0.3));
ws_3.ops.add(pcv_2);
ws_3.lambdas.add(new Double(0.7));
PropertyConditionValue pcv_6 = new PropertyConditionValue();
pcv_4.pcName = "director";
Mul mul_5 = new Mul();
mul_5.op1 = ws_3;
mul_5.op2 = pcv_6;
movieSet.connectingStrategy = mul_5;
```

4.2.4 Unscharfe Typvergleichsstrategien

Für die Typvergleichsstrategie existieren allgemeinere, unscharfe Formen, die den Abstand der zu vergleichenden Typen in Betracht ziehen. Folgende Varianten stehen zur Verfügung:

- `super[`n`,`f`]`. Elemente der Menge dürfen auch von einer Oberklassen der Mengenklasse instanziiert sein, maximal jedoch über n Vererbungsbeziehungen, sonst ist der Zugehörigkeitswert 0. Für eine exakte Typübereinstimmung ergibt sich ein Zugehörigkeitswert von 1, für jede dazwischenliegende Beziehung wird der Zugehörigkeitswert durch eine Multiplikation mit $f \in [0, 1]$ abgeschwächt.

- `super[`n`]`. Gilt als Abkürzung für `super[`n`, 0.5]`.

In j-dsd wird eine unscharfe Typvergleichsstrategie wie gesehen direkt in der rechten unteren Ecke des Kästchens notiert. In j-dsd erfolgt ein Eintrag im Javaattribut `typeCheckStrategy` von `Set`. f-dsd verwendet die Schreibweise `or supertype[n, f]` bzw. `or supertype[n]` .

4.2.5 Test auf unscharfe Mengenzugehörigkeit

Wie beim Test auf scharfe Mengenzugehörigkeit werden die Ergebnisse der Einzelbedingungen (Typ-, direkte und Attributbedingungen) in jedem Fall konjunktiv, d.h. im unscharfen Fall multiplikativ verrechnet. Zur Menge gehören genau die Instanzen, deren so errechneter Zugehörigkeitswert > 0 ist. Der Wert gibt den Grad der Zugehörigkeit an. Instanzen mit einer Zugehörigkeit von 0 gelten als nicht zur Menge gehörig.

In j-dsd übernimmt die bereits in Abschnitt 4.1.7 vorgestellte Methode `contains` die Berechnung der Zugehörigkeit. Als Rückgabewert liefert sie `double`.

5 Definieren von Variablen

Variablen stellen eine eine besondere Art von Mengen dar. Auch sie werden an Stellen einer Angebots- oder Anfragebeschreibung eingesetzt, an denen es zum Zeitpunkt der Beschreibungserstellung nicht möglich ist, einen konkreten Wert bzw. eine konkrete Instanz anzugeben. Im Unterschied zu Mengen muss dieser Wert jedoch im Verlauf der Dienstnutzung, d.h. vor oder nach der Dienstausführung, vom Dienstnehmer bzw. Dienstgeber eindeutig spezifiziert werden. Der Wert, mit dem die Variable belegt wird, muss jedoch in jedem Fall aus der Menge (der so genannten *Grundmenge*) stammen. Eine solche *ausgefüllte* Variable gilt dann als einelementige Menge, die genau diesen Wert enthält.

5.1 Variablenkategorie

Variablen werden in Kategorien eingeteilt, je nachdem von wen und wann sie ausgefüllt werden müssen. Generell unterscheidet man zwei Arten: IN-Variablen und OUT-Variablen:

- **IN-Variable.** Diese muss im Verlauf der Dienstnutzung vom Dienstnehmer ausgefüllt werden. IN-Variablen in der Anfrage (so genannte ReqIN-Variablen) sind dann sinnvoll, wenn der Dienstnehmer die Anfrage häufiger in unterschiedlicher Parametrisierung ausführen will. Vor dem Absenden der Anfrage müssen jedoch alle ReqIN-Variablen ausgefüllt werden. IN-Variablen im Angebot (so genannte OffIN-Variablen) müssen vom Dienstnehmer ausgefüllt werden, bevor der Dienst gestartet werden kann, um ihn zu konfigurieren.
- **OUT-Variable.** Diese muss im Verlauf der Dienstnutzung vom Dienstgeber ausgefüllt werden. OUT-Variablen in der Anfrage (so genannte ReqOUT-Variablen) stehen für Informationen über die Dienstausführung, an denen der Anfrager interessiert ist. Diese werden während der Schätzphase (siehe unten) oder nach der Dienstausführung eingetragen. OUT-Variablen im Angebot (so genannte OffOUT-Variablen) stehen für Informationen, die der Dienstanbieter auf Nachfrage bzw. nach Dienstausführung bekannt gibt.

Für Variablen in der Angebotsbeschreibung kann weiterhin angegeben werden, wann sie ausgefüllt werden müssen. Man unterscheidet hier eine *Schätzphase* und eine *Ausführungsphase*.

- Die **Schätzphase** läuft vor der eigentlichen Dienstausführung ab und ermöglicht dem Dienstnehmer, sich über Details des Dienstes zu informieren, die nicht direkt in der Angebotsbeschreibung festgehalten werden können. Hierzu zählen aktuelle Informationen (etwa die derzeitige Auslastung eines Druckdienstes) oder Informationen, die von Einstellungen des Dienstnehmers abhängen (wie etwa die Kosten der Inanspruchnahme eines Druckdienstes für einen farbigen Druck in 600dpi). Variablen, die in der Schätzphase auszufüllen sind, werden mit dem Index e (für estimate, engl. schätzen) gekennzeichnet. Eine zusätzlich Angabe einer Schrittnummer i gibt an, welche IN_e- und OUT_e-Variablen zusammengehören. Generell gilt: Um die Informationen der $\mathsf{OUT}_{e,i}$-Variable vom Dienstgeber zu erhalten, müssen zunächst vom Dienstnehmer alle $\mathsf{IN}_{e,i}$-Variablen spezifiziert werden.
- Die **Ausführungsphase** bezeichnet die Phase der Dienstnutzung, in welcher der Effekt des Dienstes vom Dienstgeber erbracht wird. Variablen, die in dieser Phase verwendet werden, tragen den Index x (für execute, engl. ausführen). Dabei gilt, dass IN_x-Variablen vor der Dienstausführung vom Dienstnehmer ausgefüllt werden müssen, während OUT_x-Variablen nach der Dienstausführung vom Dienstgeber bekannt gegeben werden. Wird kein Index gegeben, so gilt die Ausführungsphase als Standard.

Eine Variable gehört immer zu mindestens einer Kategorie, kann jedoch auch zu mehreren Kategorien gehören, etwa wenn ein Wert sowohl in der Schätz- als auch in der Ausführungsphase benötigt wird.

In f-dsd werden Variablen durch `var (IN|OUT, e|x,` i`)+ from <Grundmenge>` definiert. Durch die drei Parameter wird also die Kategorie der Variable festgelegt. Dabei sollte i auf 1 gesetzt werden, wenn eine Variable der Ausführungsphase definiert wird (d.h. der zweite Parameter ist x). Solche Kategorieangaben können beliebig oft wiederholt werden (gekennzeichnet durch das +). `<Grundmenge>` bezeichnet den Wertebereich der Variable; der Variable kann nur ein Wert bzw. eine Instanz dieser Menge zugewiesen werden. Die Grundmenge kann durch eine Klasse oder eine deklarative Menge nach Kapitel 4 angegeben werden. Beispiele für Variablendefinitionen in f-dsd könnten sein:

```
var (in,x,1) from Date
```

Semantik: Zur Ausführung des Dienstes muss der Dienstnehmer einen Datumswert spezifizieren.

```
var (out,x,1) from
(
   set of Cinema
   where address in
   (
       set of Address
       where city == karlsruhe
   )
)
```

Semantik: Nach der Dienstausführung gibt der Dienstgeber eine Kinoinstanz zurück (etwa das Kino, in dem eine Karte reserviert wurde). Dieses Kino hat auf jeden Fall eine Adresse in Karlsruhe.

```
var (in,e,4) (in,x,1) from Resolution
var (in,e,4) (in,x,1) from Quality
```

```
var (out,e,4) from Price
var (out,e,4) from
(
   set of DateTime
   with elements < <2004-08-31T18:00>
)
```

Semantik: Nach Angabe einer Auflösungs- (Resolution) und einer Qualitätsinstanz durch den Dienstnehmer, gibt der Dienstgeber einen Zeitpunkt und einen Preis zurück (etwa den Fertigstellungszeitpunkt sowie die Kosten des verlangten Ausdrucks). Der Zeitpunkt liegt auf jeden Fall vor 18:00 Uhr am 31.08.2004. Die Auflösung und die Qualität sind auch für die Ausführung des Dienstes relevant.

In g-dsd werden Variablen ähnlich wie Mengen dargestellt: Als Kästchen mit einem Querstrich in der linken oberen Ecke. Zur Unterscheidung werden Variablen hellgrau hinterlegt. Die Kategorie der Variable wird in die linke obere Ecke eingetragen. Hierbei sind Abkürzungen möglich, die durch x oder 1 ersetzt werden. Die Grundmenge ist die durch die Bedingungen und Strategien an diesem Kästchen definierte Menge. Die Beispiele in g-dsd zeigt Abbildung 28.

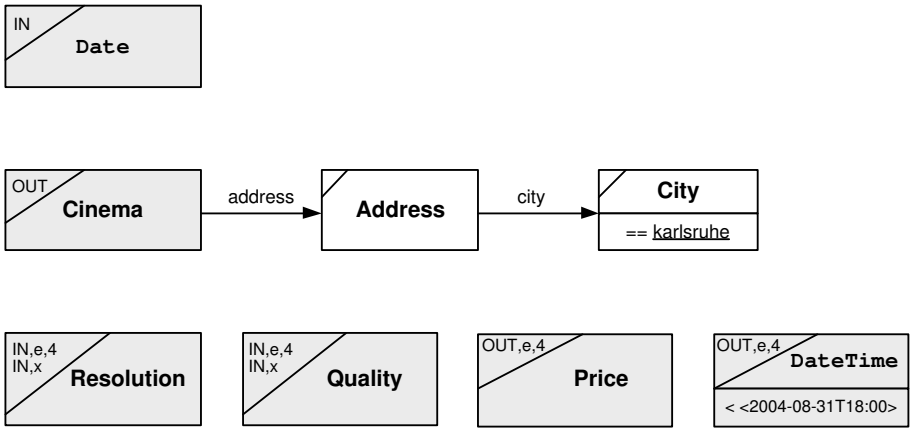


Abbildung 28: Beispiele für Variablen in g-dsd.

Da Variablen spezielle Mengen sind, existiert in j-dsd eine zusätzliche Java-Klasse `Variable` in `dsd.elements`, die von der Klasse `Set` erbt. Variablen werden dann wie Sets durch Instanzen dargestellt, in denen einerseits wie gehabt das Attribut `isSet` auf `true`, andererseits das neue Attribut `bindingStatus` auf `OPEN` gesetzt wird. Standardmäßig steht dieser `bindingStatus` auf `NO_VARIABLE`, was angibt, dass es sich um eine normale Instanz bzw. eine normale Menge handelt. Mehr zum Bindungszustand einer Variable findet sich im nächsten Abschnitt. Die Kategorie (bzw. die Kategorien) einer Variable werden mit dem Attribut `cats` beschrieben. Es nimmt einen Vector vom Typ `VariableCategory` auf. Auch diese Klasse ist in `dsd.elements` definiert und hat drei Attribute: `where`, welches mit `IN` oder `OUT` belegt werden muss, `when`, welches mit `E` oder `X` belegt werden muss, und `step`, welches mit der Schrittnummer i belegt wird. In j-dsd besteht zudem die Möglichkeit, Variablen mit einem internen Namen zu versehen. Dies ist jedoch nur für eine korrekte Zuordnung beim Dienstgeber nötig.

Die zwei ersten Beispiele von oben sehen in j-dsd wie folgt aus:

```
XSD_Date dateVar = new XSD_Date();
dateVar.isSet = true;
```

```
dateVar.setBindingStatus(Variable.OPEN);
dateVar.getCats().add(new VariableCategory("IN","X","1"));

Cinema cinemaVar = new Cinema();
cinemaVar.isSet = true;
cinemaVar.setBindingStatus(Variable.OPEN);
cinemaVar.getCats().add(new VariableCategory("OUT","X","1"));
Address addressSet = new Address();
addressSet.isSet = true;
addressSet.city = ...
cinemaVar.address = addressSet;
```

5.2 Bindungszustand

Der *Bindungszustand* einer Variable gibt an, ob und wie die Variable mit einem konkreten Wert gefüllt ist. Man unterscheidet vier Zustände (siehe Abbildung 29):

- **OPEN.** Die Variable ist noch ungebunden, das heißt sie wurde noch mit keinem Wert gefüllt.
- **BOUND.** Bezeichnet das Gegenteil von `OPEN`. Die Variable hat bereits einen Wert zugewiesen bekommen. Dieser Zustand ist eine abstrakte Oberklasse von `FILLED` und `CONNECTED` und tritt selbst nicht auf.
- **FILLED.** Eine Möglichkeit von `BOUND`. Bezeichnet eine Variable, der ein konkreter, bekannter Wert bzw. eine konkrete, vollständig bekannte Instanz (benamt oder anonym) zugewiesen wurde.
- **CONNECTED.** Eine Möglichkeit von `BOUND`. Bezeichnet eine Variable, die mit einem noch unbekannten Wert gefüllt wurde, d.h. diesen Wert übernimmt, sobald er feststeht. Meist werden ReqOUT-Variablen direkt mit OffOUT-Variablen verbunden.

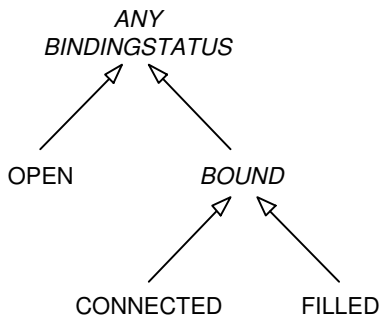


Abbildung 29: Bindungszustände von Variablen.

In f-dsd und g-dsd können Bindungszustände nicht angegeben werden. Hier gelten alle Variablen als ungebunden. In j-dsd wird der Bindungszustand durch das Javaattribut `bindingStatus` in `Variable` beschrieben. Hierzu stehen die vier vorgestellten Zustände als Konstanten zur Verfügung. Ist der Zustand `FILLED`, so muss außerdem der zugewiesene Wert im Attribut `boundValue` abgelegt werden. Dies geschieht mit der Methode `fillWith`. Im Falle von `CONNECTED` steht hier eine Referenz auf den unvollständigen Wert (meist eine verbundenen OffOUT-Variable). Dieser kann mit der Methode `connectTo` eingefügt werden. Ein Beispiel könnte sein:

```
XSD_Date dateVar = new XSD_Date();
dateVar.isSet = true;
dateVar.getCats().add(new VariableCategory("IN","X","1"));
dateVar.setBindingStatus(Variable.FILLED);
dateVar.fillWith(new XSD_Date("2004-08-31"));
```

5.3 Standardwert

Eine Besonderheit von OffIN-Variablen sind *Standardwerte*. Diese können vom Dienstanbieter angegeben werden, um Vorschläge zu machen, welche Werte zur Belegung der Variable günstig sind. Sie stellen daher Vorschläge des Dienstgebers dar, die im Zweifelsfall verwendet werden können. Beispielsweise könnte der Anbieter eines Druckdienst für die IN-Variable der Auflösung einen Wert von 600dpi vorschlagen. Diese sollte vom Vergleicher verwendet werden, wenn der Dienstnehmer in seiner Anfrage keine Präferenzen für die Auflösung angegeben hat. Wichtig: Der Defaultwert selbst muss immer zur Grundmenge der Variable gehören.

In f-dsd wird ein Standardwert durch das Schlüsselwort `default <Standardwert>` notiert. Die Angabe erfolgt am Ende der Variablendefinition. Drei Beispiele:

```
var (in,x,1) from Date default <2005-01-01>

var (in,x,1) from Movie default harryPotter3

var (in,x,1) from Price
default anonymous Price
        [
          amount = 2.00,
          currency = eur
        ]
```

In g-dsd wird der Standardwert durch einen speziellen, gestrichelten Pfeil mit der kursiven Beschriftung default angehängt. Die Beispiele von oben in g-dsd zeigt Abbildung 30.

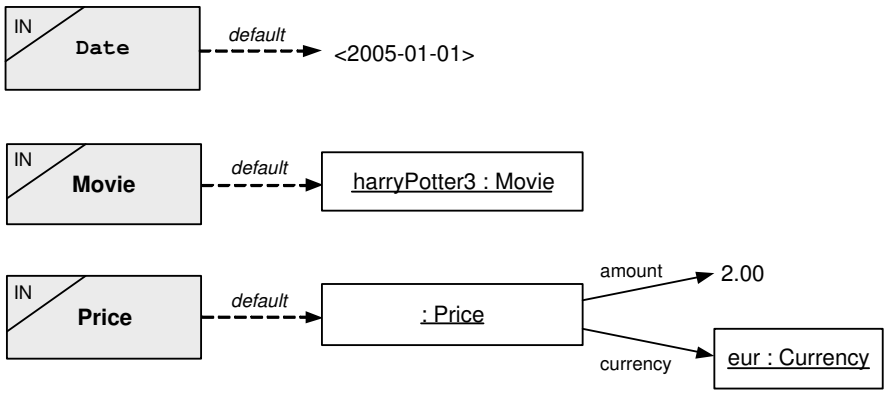


Abbildung 30: Beispiele für Standardwerte bei OffIN-Variablen in g-dsd.

In j-dsd existiert ein Javaattribut `defaultValue` in `Variable`, in das der optionale Standardwert eingetragen werden kann. Das erste Beispiel sieht also in j-dsd wie folgt aus:

```
XSD_Date dateVar = XSD_Date();
dateVar.isSet = true;
dateVar.setBindingStatus(Variable.OPEN);
dateVar.getCats().add(new VariableCategory("IN","X","1"));
dateVar.setDefaultValue(new XSD_Date("2005-01-01"));
```

Teil III

Beschreiben von Diensten mit DSD

Im Folgenden wird erläutert, wie die Beschreibungselemente aus dem letzten Teil verwendet werden können, um Dienstangebote und -anfragen zu beschreiben.

6 Grundlagen zu Dienstbeschreibungen

Dieses Kapitel untersucht grundlegende Eigenschaften von Dienstbeschreibungen: ihre Ziele, ihren Lebenszyklus während einer Dienstnutzung, ihren Aufbau durch Mischen von Instanzen, Mengen und Variablen sowie ihre Grundstruktur.

6.1 Ziel von Dienstbeschreibungen

Die Ziele einer Dienstbeschreibung unterscheiden sich für Dienstangebots- und -anfragebeschreibungen.

6.1.1 Angebotsbeschreibungen

Eine Angebotsbeschreibung drückt aus, welchen Effekt (oder welche Effekte) der angebotene Dienst erbringen kann. In der Regel geschieht dies durch zwei Teile: Die Beschreibung des Objekts, auf dem der Dienst operiert, sowie der Beschreibung des neuen Zustands, der für dieses Objekt im Rahmen der Dienstausführung erzielt wird. Beispiele für Effekte könnten sein: eine Kinokarte ist gebucht, ein Dokument ist gedruckt, eine PDF-Datei wird lokal verfügbar gemacht, eine Aktie wird bzgl. ihres Kurses bekannt gemacht, ein Auto wird gemietet, ein Buch wird gekauft.

In der Regel kann ein angebotener Dienst nicht nur genau einen Effekt erbringen, sondern kann mit verschiedenen ähnlichen Objekten umgehen und diese in verschiedene ähnliche Zustände überführen. Insgesamt ist also in einer Dienstangebotsbeschreibung eine *Familie von Effekten* zu erfassen. Aus diesem Grund ist es nötig, Mengen in die Beschreibung des Dienstes aufzunehmen, um so alle vom Dienst erzielbaren Effekte zu erfassen. Hierbei ist zu beachten, dass der Dienst bei seiner Ausführung nur einen einzigen dieser Effekte tatsächlich erbringt. Durch Einbringen von IN-Variablen erhält der Dienstnehmer die Möglichkeit, den tatsächlich erbrachten Dienst eindeutig zu bestimmen oder zumindest auf eine für ihn geeignete Menge einzuengen. OUT-Variablen helfen dem Dienstnehmer, Informationen über den erbrachten Effekt zu erhalten, um ihn so eindeutig zu kennen oder zumindest nützliche Informationen für eine Nachverarbeitung zu erhalten.

6.1.2 Anfragebeschreibungen

Eine Anfragebeschreibung muss erfassen, an welchen Effekten der Dienstnehmer interessiert ist. Wie schon in der Einführung erläutert, versucht der Dienstnehmer in der Regel, eine bestimmte Aufgabe zu erledigen und toleriert daher eine Reihe von Effekten mit unterschiedlicher Präferenz. Im Gegensatz zu Angebotsbeschreibungen werden in Anfragebeschreibungen daher Mengen nicht punktuell eingesetzt, sondern bereits die Beschreibung des Effekts liegt als Menge vor. Sie enthält alle die Effekte, die der Dienstnehmer bereit ist, für seine Aufgabe zu akzeptieren. Zudem ist es

durch die Angabe unscharfer Mengenzugehörigkeiten möglich, Präferenzen unter den Elementen festzulegen.

Auch Variablen haben in Anfragebeschreibungen einen anderen Zweck. IN-Variablen helfen, die Anfrage für verschiedene, ähnliche Situationen wiederzuverwenden. Dazu kann die Anfrage bei jeder erneuten Ausführung konkret konfiguriert werden. OUT-Variablen hingegen ermöglichen es dem Dienstnehmer anzugeben, welche Informationen er über den erbrachten Effekt benötigt, um diesen angemessen nutzen zu können.

Ausgehend von einer solchen Anfragebeschreibung muss im Rahmen der Dienstnutzung ein geeignetes Angebot gefunden werden, dessen Effekt (bestehend aus bearbeitetem Objekt und Zustandsänderung) mittels der OffIN-Variablen so spezifiziert oder zumindest eingeengt werden kann, dass ein vom Dienstnehmer möglichst stark präferierter Effekt entsteht. Zudem muss der Dienstnehmer am Ende der Dienstnutzung über den tatsächlich erbrachten Effekt mit all den Informationen versorgt werden, die er in Form von ReqOUT-Variablen spezifiziert hat.

6.2 Lebenszyklus einer Dienstbeschreibung während der Dienstnutzung

Um die Semantik von Variablen in Dienstbeschreibungen verstehen zu können, ist es nötig, sich den Lebenszyklus von Beschreibungen während der Dienstnutzung genauer anzusehen (siehe auch [11]).

Vor der eigentlichen Dienstnutzung steht die **Registrierungsphase**. Hier legt der Dienstnehmer eine in Zukunft häufiger benötigte Dienstanfrage unter einem Namen ab. Diese Dienstanfrage enthält für gewöhnlich sowohl IN- als auch OUT-Variablen.

Die eigentliche *Dienstnutzung* findet in drei Phasen statt: die Such-, die Schätz- und die Ausführungsphase (siehe Abbildung 31):

- In der **Suchphase** ruft der Dienstnehmer einen der zuvor registrierten Dienste unter seinem Namen auf, füllt dazu die IN-Variablen in der hinterlegten Anfragebeschreibung mit konkreten Werten und sendet sie an die Dienstvermittlung. Diese sucht mittels eines Vorvergleichs Beschreibungen von Dienstangeboten heraus, die möglicherweise passen könnten und liefert sie an den Dienstnehmer zurück. Diese können IN_e-, IN_x-, OUT_e- und OUT_x-Variablen enthalten.
- In der **Schätzphase** wertet der Dienstnehmer mittels des Vergleichs die bereitgestellten Dienstangebote aus. Hierzu fragt er gegebenenfalls direkt bei den Dienstgebern nach konkreten $\mathsf{OUT}_{e,i}$-Werten nach, indem er ihnen die jeweils benötigten $\mathsf{IN}_{e,i}$-Werte sendet. Der Vergleicher hat mehrere Aufgaben: Er muss das beste zur Anfrage passende Angebot auswählen, die OffIN-Variablen füllen und die ReqOUT-Variablen binden (d.h. füllen oder verbinden).
- In der **Ausführungsphase** stößt der Dienstnehmer den ausgewählten Dienst beim entsprechenden Dienstnehmer an und sendet ihm die gefüllten OffIN-Variablen. Der Dienstgeber führt daraufhin den Dienst aus, d.h. erbringt einen der beschriebenen Effekte. Anschließend sendet er Informationen hierüber in Form gefüllter OffOUT-Variablen an den Dienstnehmer zurück. Dieser schließt die Dienstausführung ab, indem er die verbundenen ReqOUT-Variablen entsprechend der gefüllten OffOUT-Variablen füllt.

6.3 Aufbau durch Mischen von Instanzen, Mengen und Variablen

Wie gesehen sind in Angebotsbeschreibungen sowohl Instanzen als auch Mengen und Variablen nötig, um einen Dienst korrekt beschreiben zu können. Diese wurden bisher nur getrennt verwendet, d.h. Instanzen hatten andere Instanzen als Füllwerte ihrer Attribute, Mengen hatten andere Mengen als Attributbedingungen. In Angebotsbeschreibungen treten sie gemischt auf. Hierbei sind zwei *Mischungsregeln* zu beachten.

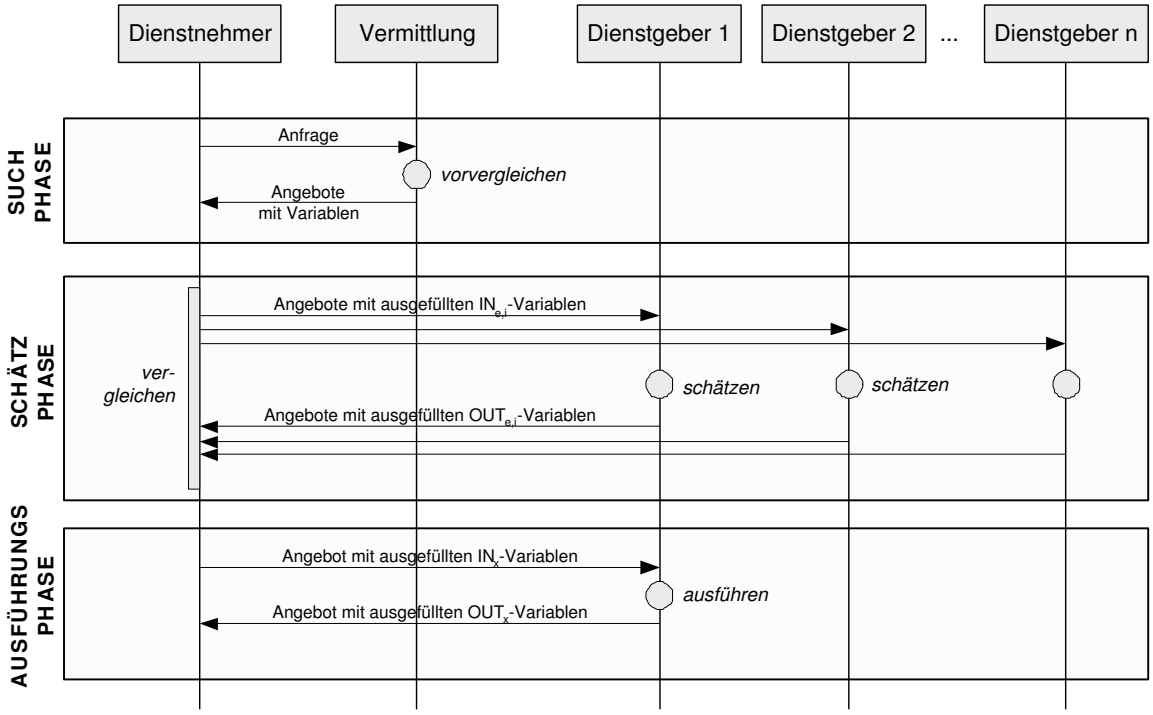


Abbildung 31: Phasen der Dienstnutzung nach [11].

6.3.1 Einbringen von Mengen

Die erste Mischungsregel gestattet das Einbringen von Mengen in Dienstbeschreibungen.

Mischungsregel 1: Einbringen von Mengen
Füllwerte von anonymen Instanzen dürfen Mengen sein. In diesem Fall muss jedes Element der Menge einen gültigen Füllwert darstellen. Andere Verbindungen zwischen Mengen und Instanzen sind nicht gestattet.

Diese Regel erlaubt es also, Attribute von anonymen Instanzen mit Mengen anstatt mit Instanzen zu füllen.

Ein Beispiel in g-dsd zeigt Abbildung 32. Hier ist in der anonymen Instanz von Price der Füllwert des Attributs amount durch eine Menge von `Double`-Werten ersetzt.

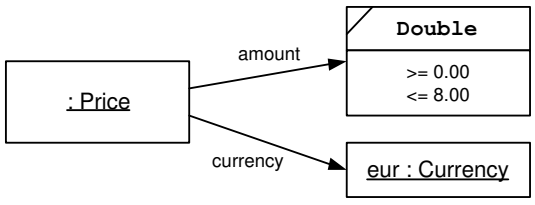


Abbildung 32: Beispiel für die Anwendung der ersten Mischungsregel in g-dsd.

Konsequenterweise wird das Beispiel wie folgt in f-dsd notiert:

```
anonymous Price
[
   amount = (
              set of Double
              with elements >= 0.00
              with elements <= 8.00
            ),
   currency = eur
]
```

In j-dsd sieht das Beispiel wie folgt aus:

```
Price p = new Price();
p.amount = new XSD_Double();
p.amount.isSet = true
p.amount.directConditions = new Vector();
p.amount.directCOnditions = ...
p.currency = dsd.instance.domain.money.Currency.eur.create();
```

Innerhalb von Angebotsbeschreibungen ergibt sich für eine solche Mischung folgende Semantik: Vor der Dienstausführung ist der durch eine Menge ersetzte Füllwert noch undefiniert. Nach der Dienstausführung ist der Füllwert eindeutig definiert und stellt ein Element der Menge dar. Zu beachten ist, dass der Dienstnehmer den gewählten Wert dennoch nicht kennen wird. Für ihn ist aber sicher, dass er aus der Menge stammt.

Zur Verdeutlichung: Die Regel lässt ausdrücklich nicht zu, dass Füllwerte von benamten Instanzen durch Mengen ersetzt werden (da diese ja bereits schon an anderer Stelle vollständig definiert wurden) oder dass Attributbedingungen von Mengen durch Instanzen ersetzt werden.

6.3.2 Einbringen von Variablen

Die zweite Mischungsregel gestattet das Einbringen von Variablen in Dienstbeschreibungen:

Mischungsregel 2: Einbringen von Variablen
Mengen dürfen durch Variablen ersetzt werden. Allerdings darf von keiner IN-Variable ein Pfad über Attributbedingungen zu einer anderen IN-Variable führen.

Diese Regel erlaubt es also, Mengen durch Variablen einer beliebigen Kategorie zu ersetzen. Dabei ist es unerheblich, ob die Menge durch die Mischungsregel 1 in die Beschreibung eingebracht wurde oder als Attributbedingung auftritt.

Ein komplexeres Beispiel für die Anwendung der Regel zeigt Abbildung 33. Hier wurden zwei Mengen durch Variablen ersetzt: eine OUT-Variable ersetzt den Füllwert der anonymen Price-Instanz, eine IN-Variable ersetzt die Menge, die als Attributbedingung der Movie-Menge diente.

Die Notation in f-dsd sieht wie folgt aus:

```
anonymous CinemaTicket
[
  price = anonymous Price
          [
```

```
            amount = var (out,x,1) from
                     (
                       set of Double
                       with elements >= 0.00
                       with elements <= 8.00
                     ),
            currency = eur
          ],
  validFor = (
              set of SeatInShow
              where visible in
              (
                set of Movie
                where genre == var (in,x,1) from
                               (
                                 set of Genre
                               )
              )
             )
]
```

Zu beachten ist die Notation `genre == var(in,x,1)`, d.h. wenn Variablen als Attributbedingungen auftreten, werden sie wie Einzelinstanzen mit dem `==`-Operator verglichen.

Die Notation in j-dsd folgt der bekannten Syntax.

Die durch Variablen ersetzten Mengen haben folgende Semantik:

- Solange sie *ungebunden* sind, wirken sie wie ihre Grundmenge, d.h. es ist nur gewiss, dass am Ende der Dienstausführung ein konkreter Wert aus ihnen ausgewählt sein wird.
- Sobald sie *gefüllt* sind (also bei OffIN-Variablen direkt vor und bei OffOUT-Variablen direkt nach der Dienstausführung), wirken sie wie eine einelementige Menge, die als einziges Element den in die Variable eingefüllten Wert enthält.

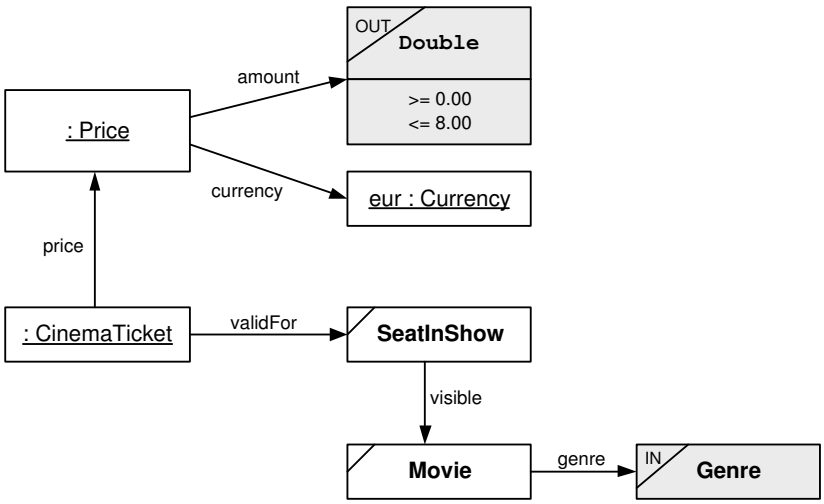


Abbildung 33: Beispiel für die Anwendung der zweiten Mischungsregel in g-dsd.

Im Beispiel geht es also um eine Kinokarte, die noch teilweise undefiniert ist. Sicher ist schon, dass sie für einen Platz in einer Kinovorstellung gültig sein und einen Preis unter 8 Euro haben wird. Das Genre des Film der Kinovorstellung kann vor der Dienstausführung vom Dienstnehmer eingefüllt werden. Hierdurch wird indirekt auch die Menge der SeatInShows eingeschränkt. Nach der Dienstausführung ist die Kinokarte vollständig spezifiziert. Dem Dienstnehmer wird zwar der Preis der Karte aufgrund der OUT-Variable mitgeteilt, die gebuchte Platznummer und Reihe sind ihm in diesem Beispiel aber nicht bekannt.

6.4 Genereller Aufbau von Dienstbeschreibungen

Durch die beiden Mischungsregeln ergibt sich eine einheitliche Struktur für Dienstbeschreibungen, die für Angebots- und Anfragebeschreibungen unterschiedlich ist.

Abbildung 34 zeigt den generellen Aufbau einer Angebotsbeschreibung. Beginnend mit der zu definierenden benamten Instanz vom Typ Service wird das Attribut presents mit einer anonymen Instanz vom Typ ServiceProfile gefüllt. Diese hat effect als Attribut, welches mit einer anonymen Instanz vom Typ einer speziellen, wertbestimmten State-Klasse gefüllt wird. Diese Instanz selbst kann als Füllwerte ihrer weitere anonyme Instanzen (rekursiver Pfeil), benamte Instanzen oder Mengen/Variablen besitzen. Benamte Instanzen sind bereits in einer Ontologie definiert und besitzen keine neue Füllungen ihrer Attribute. Mengen bzw. Variablen können nach Mischungsregel 2 gleichartig verwendet werden. Ihre Attributbedingungen können nur auf weitere Mengen bzw. Variablen verweisen.

Anfragebeschreibungen sind einfacher aufgebaut (siehe Abbildung 35). Auch hier beginnt die Beschreibung mit einer benamten Instanz vom Typ Service, die als Füllwert im Attribut presents eine anonyme Instanz vom Typ ServiceProfile besitzt. Der Füllwert an effect ist jedoch direkt eine Menge (oder Variable), da der Dienstnehmer seine Aufgabe durch unterschiedliche Effekte erledigen kann. Die Mengen/Variablen sind in der Regel unscharf, um die Präferenzen auszudrücken. Ihre Attributbedingungen verweisen auf weitere unscharfe Mengen bzw. Variablen.

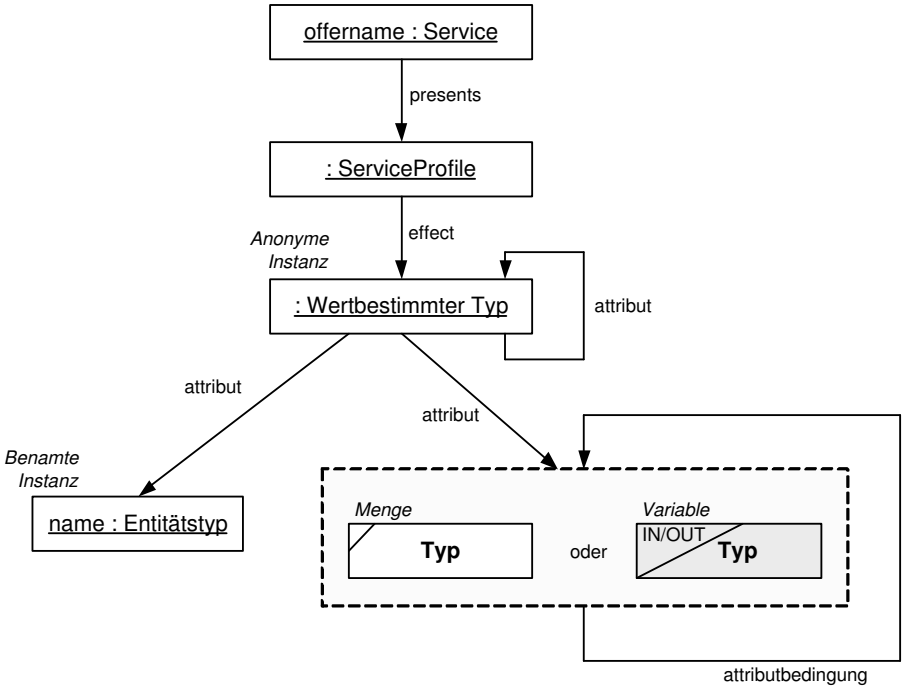


Abbildung 34: Genereller Aufbau von Angebotsbeschreibungen.

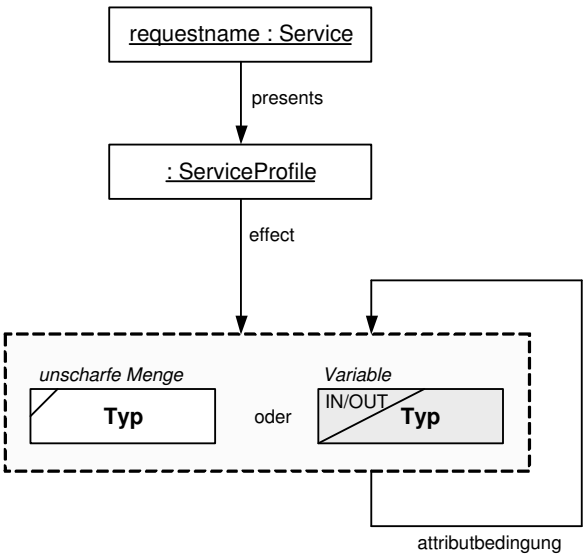


Abbildung 35: Genereller Aufbau von Anfragebeschreibungen.

7 Allgemeiner Prozess zum Aufbau von Dienstbeschreibungen

Da eine Dienstbeschreibung im Wesentlichen eine Instanz der Klasse Service darstellt, ist die Grundtechnik zur Erstellung einer Dienstbeschreibung die Instanziierung. Für einen späteren Vergleich ist es dabei wichtig, dass die Beschreibung einen relativ strukturierten Aufbau hat. Um dies zu erreichen, folgt die Instanziierung einem definierten Ablauf, der im Wesentlichen dadurch entsteht, dass Ontologien geschichtet und fortlaufend instanziiert werden (siehe auch [7]). Wie bereits in Abschnitt 1.1.3 kurz vorgestellt, existieren drei Schichtungsstufen: die obere Dienstontologie, die Kategorieontologien sowie die Domänenontologien. Im Folgenden werden diese Stufen im Detail vorgestellt.

7.1 Obere Dienstontologie

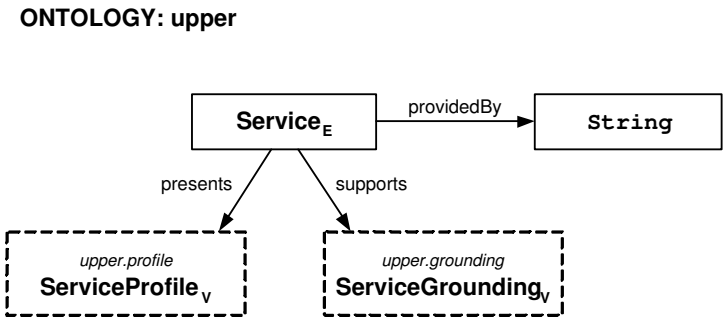


Abbildung 36: Obere Dienstontologie in g-dsd.

Die obere Dienstontologie ist ein Schema, welches das allgemeine Grundgerüst für Dienstbeschreibungen festlegt. Abbildung 36 zeigt diese Ontologie `upper` in g-dsd. Das Schema ist sehr einfach und übernimmt im Wesentlichen die Idee aus OWL-S [17], die unterschiedlichen Aspekte eines Dienstes getrennt zu beschreiben. In DSD erfolgt das in zwei Teilen: Das ServiceProfile enthält eine abstrakte Blackbox-Beschreibung, was der Dienst macht. Im ServiceGrounding ist festgehalten, wie die Verbindung vom abstrakten zum realen Dienst aussieht. Zudem wird durch das Attribut providedBy der Dienstanbieter erfasst. In TCP/IP-basierten Netzen ist das für gewöhnlich eine IP-Adresse und eine Portnummer, z.B. `"192.168.0.1:8088"`. Die Details zum Grounding finden sich in Abschnitt 10.

Abbildung 37 zeigt die (vereinfachte) Darstellung des Schemas für das ServiceProfile in g-dsd.[11] Es realisiert das Prinzip rein zustandsorientierter Dienstbeschreibungen, indem es zur Beschreibung von Diensten nur zwei funktionale Attribute erlaubt:

- precondition (Vorbedingung): Beschreibt den Zustand (State) der Welt, der nötig ist, damit der Dienst erfolgreich ausgeführt werden kann. Wird der Dienst aufgerufen, obwohl die Bedingung nicht erfüllt ist, so ist der Effekt undefiniert. Neben den direkt unter precondition angegebenen Vorbedingungen, können auch noch implizite existieren. Die Details finden sich in Abschnitt 9.
- effect (Effekt): Beschreibt den Zustand der Welt nach einer *erfolgreichen* Durchführung des Dienstes. Über den Zustand im Falle einer fehlerhaften Ausführung wird ausdrücklich nichts ausgesagt, da dies bei der Suche nach einem geeigneten Dienst im Allgemeinen keinen Vorteil bringt.

[11] Das komplette Schema findet sich in Anhang A.2.

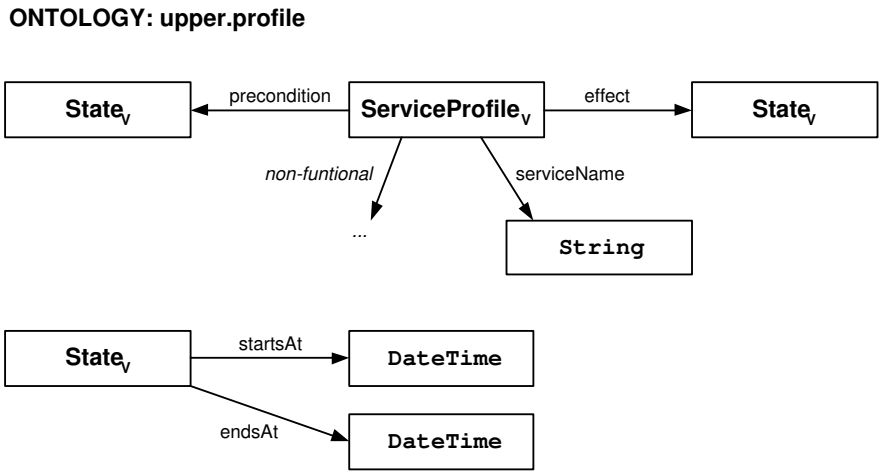


Abbildung 37: Verkürzte Darstellung der Ontologie `upper.profile`. Die vollständige Ontologie findet sich in Anhang A.2.

Eigentlich sind die Attribute precondition und effect listenwertig (siehe Anhang A.2), um mehrere Vorbedingungen oder Effekte eines Dienstes erfassen zu können.

Neben den funktionalen Attributen erlaubt das Profile auch die Beschreibung der nichtfunktionalen Aspekte. Besonders zu erwähnen ist hierbei der Name des Dienstes als `String`, welcher im Wesentlichen für administrative Zwecke verwendet wird. Spezielle nicht-funktionale Attribute sind noch nicht vorgegeben, sondern sollen in Zukunft definiert werden.

Eine wichtige Klasse des Profiles ist State, welche stellvertretend für Zustände der realen Welt steht. Die Klasse muss als abstrakt angesehen werden, da sie selbst hat keine direkten Instanzen hat. Vielmehr existieren im Rahmen der Kategorieontologien (siehe nächster Abschnitt) eine Reihe von Unterklassen von State, die instanziiert werden können. Eine solche Instanz beschreibt nie alle Aspekte der Welt vollständig. Vielmehr wird nur der interessierende Ausschnitt beschrieben, während für die restliche Welt angenommen wird, dass der Zustand ohne Belang und daher undefiniert ist. Darüberhinaus verfügt die Klasse State über zwei (optionale) Attribute startsAt und endsAt, mit deren Hilfe festgelegt werden kann, von wann und/oder bis wann der Zustand gilt.

7.2 Kategorieontologien

Die zweite Schicht von Ontologien zur Beschreibung von Diensten stellen Kategorieontologien dar. Diese teilen die Menge aller Dienste so in Gruppen auf, dass diese in ihren erzielten Zustandsübergängen wesentlich voneinander verschieden sind. Wichtigste Aufgabe der Kategorieontologien ist es daher, die abstrakte Klasse State für die verschiedenen Dienstkategorien zu konkretisieren. Ontologien aus dieser Schicht beginnen mit dem Präfix `category`.

DSD unterscheidet vier Dienstkategorien: Wissensdienste, Informationsdienste, Realweltdienste und Befähigungsdienste.

7.2.1 Wissensdienste

Wissensdienste (engl. knowledge services) dienen dazu, neue Informationen über ein Objekt zu erlangen (siehe Abbildung 38). Hierzu existiert der Zustand Known als Unterklasse von State, der

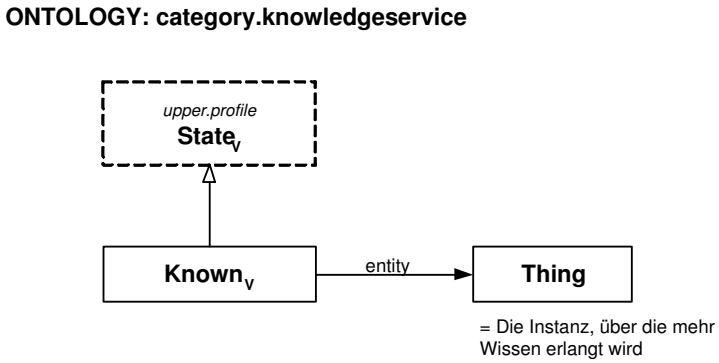


Abbildung 38: Ontologie `category.knowledgeservice` zur Darstellung von Diensten zur Erlangung von Wissen über die Füllwerte von Instanzen.

andeutet, dass Wissen über die Füllwerte einer Instanz dazugewonnen wird. Diesen Wissenszuwachs erzielt der Dienst über OUT_x-Variablen, die dem Dienstnehmer nach Ausführung des Dienstes gefüllt zur Verfügung stehen. Der Effekt des Dienstes wird also innerhalb des Dienst-Frameworks erwirkt. Das Attribut entity von Known hat als Typ die generische Klasse Thing. Instanzen von Known sollten für entity die Instanz als Füllwert haben, über die weiteres Wissen erlangt wird. Sinnvoll sind Wissensdienste nur für Instanzen von Entitätsklassen, da nur dort die Füllwerte einer Instanz untereinander abhängen können.

7.2.2 Informationsdienste

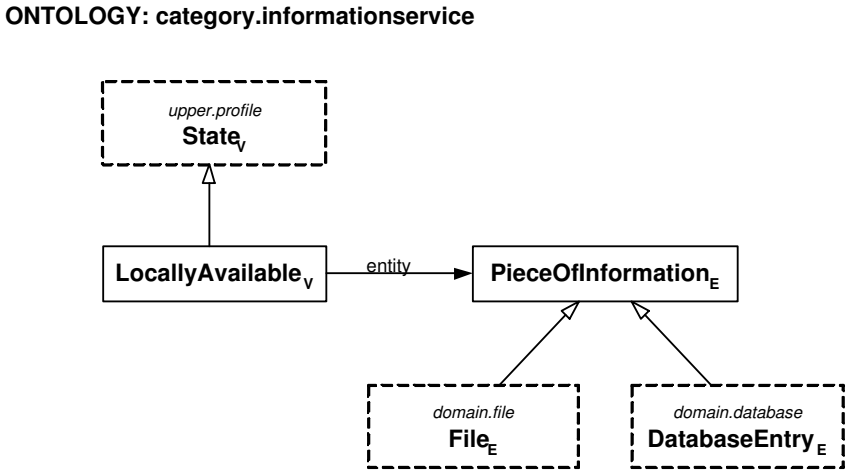


Abbildung 39: Ontologie `category.informationservice` zur Darstellung von Diensten zur Veränderung des Zustands von Informationssystemen.

Informationsdienste (engl. information services) dienen dazu, den Zustand des Informationssystems zu verändern (siehe Abbildung 39). Der Zustand eines Informationssystems ist im Wesentlichen durch die Daten bestimmt, die in ihm in Form von Dateien oder Datenbankeinträgen abgelegt sind. Als konkreter Zustand dient dabei LocallyAvailable als Unterklasse von State, der ausdrückt, dass ein Datum lokal verfügbar ist und durch das System verwendet werden kann. Steht LocallyAvailable

als Zustand der Vorbedingung, ist das System des Dienstgebers gemeint, im Effekt das des Dienstnehmers. Das Attribut entity zeigt daher auf ein allgemeines PieceOfInformation, das eine Datei (File, siehe nächster Abschnitt) oder ein Datenbankeintrag (DatabaseEntry) sein kann. Den Effekt erzielt ein solcher Dienst durch ein externes Austauschprotokoll für Dateien (wie FTP oder E-Mail) oder Datenbankeinträge (wie JDBC). Hierzu kann ein spezielles Grounding angegeben werden (siehe Abschnitt 10.5).

Neben LocallyAvailable sind noch weitere Zustände von PieceOfInformation denkbar. Diese können bei Bedarf erweitert werden.

Für Informationsdienste, die sich auf Dateien beziehen, stellt File eine wichtige Klasse dar, welche in der Ontologie `domain.file` definiert wird. In Abschnitt 7.3 wird genauer darauf eingegangen.

7.2.3 Realweltdienste

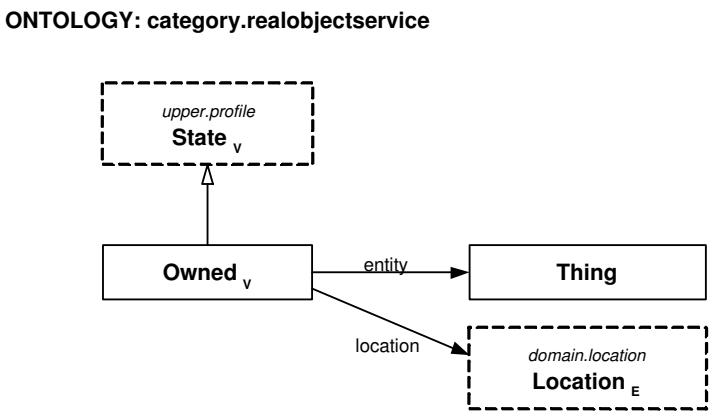


Abbildung 40: Ontologie `category.realobjectservice` zur Darstellung von Diensten zur Zustandsänderung von Objekten der realen Welt.

Realweltdienste (engl. real object services) dienen dazu, den Zustand von Objekten der realen Welt zu verändern (siehe Abbildung 40). Wichtigster Zustand ist Owned als Unterklasse von State. Dieser drückt aus, dass ein realweltliches Objekt erschaffen bzw. beschafft wird, um anschließend einem Besitzer zu gehören. Steht Owned als Zustand der Vorbedingung muss der Dienstgeber der Besitzer sein, im Effekt wird es der Dienstnehmer sein. Den Effekt erzielt ein solcher Dienst außerhalb des Dienstframeworks, etwa durch Auslösen bestimmter Realweltaktionen durch Nachrichten an Personen oder Nutzung externer Geräte wie Drucker.

Neben Owned sind noch andere Zustände realweltlicher Objekte denkbar. Diese können bei Bedarf erweitert werden.

7.2.4 Befähigungsdienste

Befähigungsdienste (engl. capability services) sind Dienste, die dem Dienstnehmer eine Fähigkeit zuteil werden lassen (siehe Abbildung 41). Meist ist diese Befähigung zeitlich begrenzt. Ein Beispiel für einen solchen Dienst ist ein Internetdienst, der es dem Dienstnehmer ermöglicht, für eine gewisse Zeit eine Verbindung zum Internet zu nutzen. Wichtigster Zustand ist Obtained als Unterklasse von State. Sein Attribut entity ist dabei eine Capability, also Fähigkeit. Typischerweise werden in Obtained die geerbten Attribute startsAt und endAt gefüllt, um die Dauer der Befähigung auszudrücken.

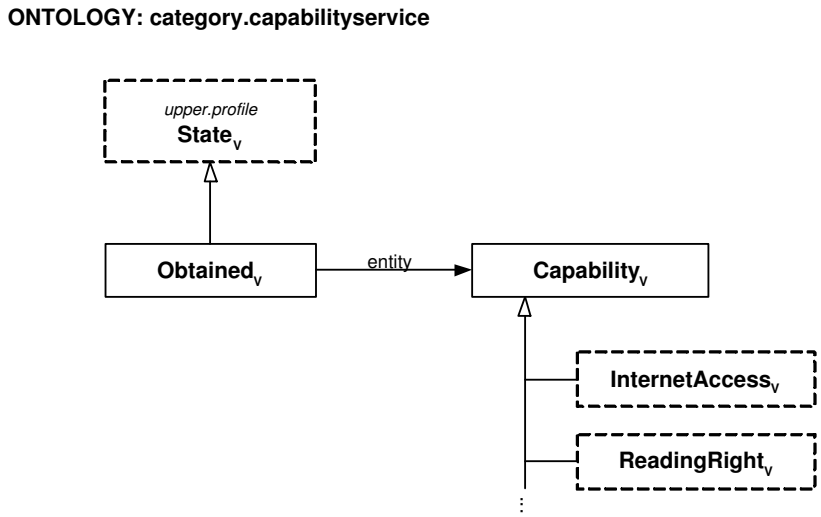


Abbildung 41: Ontologie `category.capabilityservice` zur Darstellung von Diensten, die den Dienstnehmer befähigen, eine Aktion durchführen zu können.

7.3 Domänenontologien

Die dritte Schicht der Ontologien stellen Domänenontologien dar. Sie konzeptualisieren ein bestimmtes Anwendungsgebiet, wie Bücher, Orte, Dateien usw. Im Gegensatz zu den wenigen und eher kleinen Ontologien auf den oberen beiden Schichten existieren viele und auch größere Domänenontologien. Sie werden nicht durch DSD bereitgestellt, sondern von Experten in der Community eingebracht und liegen daher im Allgemeinen verteilt vor. In der Regel enthalten Domänenontologien sowohl Klassen, die das Anwendungsgebiet inhaltlich strukturieren, als auch Instanzen, welche für konkrete, reale Individuen aus diesem Bereich stehen. Zusätzlich können domänenspezifische Vergleichsfunktionen (siehe Abschnitt 3.9) eingebracht werden, falls diese von allgemeinem Interesse sind. Domänenontologien beginnen mit dem Präfix `domain`.

Bestimmte Domänenontologien sind von großem allgemeinem Interesse und werden häufig in verschiedensten Dienstbeschreibungen verwendet. Hierunter fallen `domain.measure` zur Beschreibung einheitenbehafteter Werte, `domain.file` zur Beschreibung von Dateien und ihren Inhalten, `domain.person` zur Beschreibung von Personen und `domain.location` zur Beschreibungen von Orten. Im Folgenden werden die ersten beiden kurz vorgestellt. Ihre Repräsentation in g-dsd findet sich auch im Anhang A.

Abbildung 42 zeigt einen Ausschnitt aus der Ontologie `domain.measure` in g-dsd. In der Ontologie sind grundsätzliche Beschreibungselemente wie einheitenbehaftete Werte (*Measure) und Maßeinheiten (*Unit) beschrieben. Werte stellen dabei eine wertbasierte Klasse dar, da alle Kombinationen aus Zahlen und Einheiten sinnvolle anonyme Instanzen ergeben. Für sie sind zudem domänenspezifische Gleichheits- (`domEquals`), Ähnlichkeits- (`domSimilar`) und Ordnungsfunktionen (`domCompareTo`) definiert, welche zwei Werte unter Berücksichtigung ihrer Einheiten vergleichen. Einheiten stellen öffentliche Entitätsklassen ohne Attribute dar. Als Instanzen sind daher verwendbaren Einheiten aufgelistet.

In Anhang A.9 ist die Ontologie `domain.file` zu sehen, welche Konzepte zur Beschreibung von Dateien und ihren Inhalten enthält (siehe auch [12]). File ist hierbei eine Unterklasse von PieceOfInformation, welche im Rahmen von Informationsdiensten verarbeitet werden. File als teilöffentliche

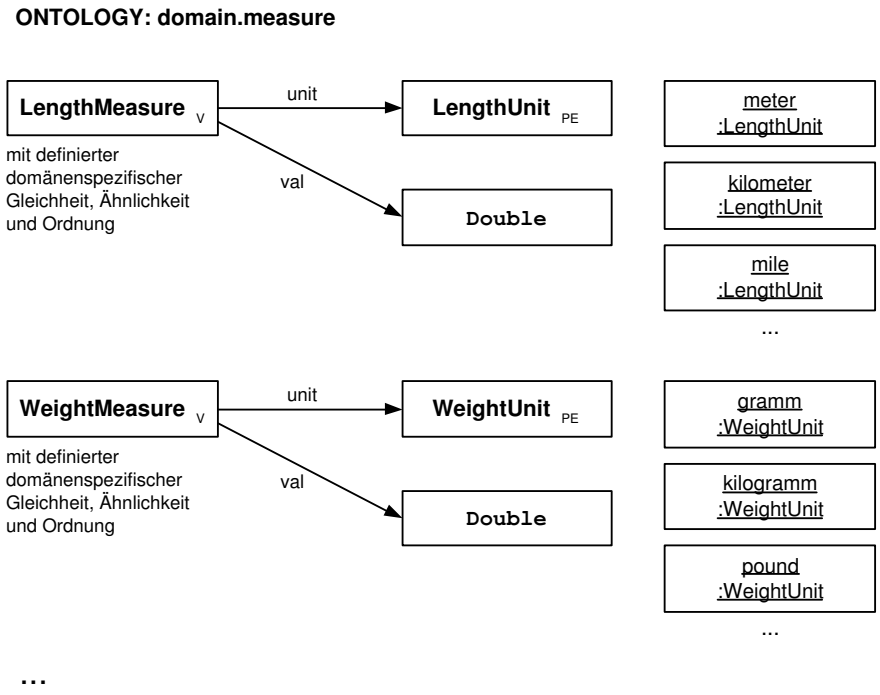


Abbildung 42: Domänenontologie `domain.measure` zur Beschreibung einheitenbehafteter Werte.

Entitätsklasse beschreibt dabei eine Datei auf technischer Ebene, d.h. mit Namen, Ersteller, Erstellungsdatum, Format, Größe usw. In bytes ist die binäre Inhalt der Datei als (langer) String hinterlegt. Auf semantischer Ebene enthält die Datei ein Dokument, was durch das Attribut contains ausgedrückt wird. Auch Document ist eine teilöffentliche Entitätsklasse und beschreibt einen Inhalt mit Titel, Autor und Dokumenttyp wie Bild, Video, Text etc. Das Thema des Dokuments wird über das Attribut dealsWith ausgedrückt. Ein solches Topic ist eine öffentliche Entitätsklasse mit Instanzen, die über isSubtopicOf hierarchisch angeordnet sind.

8 Eigenschaften von Angebots- und Anfragebeschreibungen

Beschreibungen von Dienstangeboten und Dienstanfragen unterscheiden sich in einigen Punkten voneinander. Dieses Kapitel beschreibt diese Unterschiede.

8.1 Angebotsbeschreibungen

8.1.1 Übersicht

Folgende Aufstellung gibt einen Überblick über die Besonderheiten von Angebotsbeschreibungen:

- *Keine Fuzzymengen.* Angebotsbeschreibungen beschreiben die Familie der Effekte, die der angebotenen Dienst erbringen kann. Dabei gilt, dass der Dienstanbieter keine Präferenzen unter diesen Effekten hat, zumindest keine, die er in der Dienstbeschreibung bekannt gibt. Mengen, die in Angebotsbeschreibungen eingebracht werden, sind daher immer scharfe Mengen. Dadurch sind insbesondere nur die beiden Operatoren `and` und `or` bzw. `add` und `mul` im Rahmen von Verbindungsstrategien zulässig (vgl. Abschnitt 4.1.5).
- *Eindeutig oder mehrdeutig spezifizierbar.* Dienstangebote unterscheiden sich darin, ob der Dienstnehmer eindeutig festlegen kann, welchen Effekt der Dienst erbringen soll oder nicht. Die Details hierzu finden sich in Abschnitt 8.1.2.
- *Unterbestimmte Beschreibungen möglich.* Insbesondere bei der Verwendung von teilöffentlichen Entitätsklassen kommt es häufig vor, dass der Dienst nicht genau das anbietet, was seine Beschreibung angibt. In solchen Fällen behilft man sich durch die Berechnung der Ausführungswahrscheinlichkeit. Die Details hierzu finden sich in Abschnitt 8.1.3.
- *Grounding verweist auf konkreten, realen Dienst.* Das Grounding verbindet die Variablen der Beschreibung mit den Ein- und Ausgabemöglichkeiten eines konkreten Diensts. Die Details zum Grounding finden sich in Abschnitt 10.
- *Mehrere Effekte.* Ein Dienst kann bei seiner Ausführung mehrere Effekte erwirken, die in Haupteffekte und Nebeneffekte unterteilt werden können. Die einzelnen Effekte werden als Zustände im listenwertigen Attribut effect von ServiceProfile angegeben.
- *Vorbedingungen.* Zur Ausführung eines angebotenen Dienste kann es nötig sein, dass bestimmte Bedingungen erfüllt sind. Diese werden unter anderem in den preconditions der Angebotsbeschreibung festgehalten. Die Details hierzu finden sich in Abschnitt 9.

8.1.2 Eindeutig vs. mehrdeutig spezifizierbare Dienste

Angebotene Dienste können in der Regel eine Familie von ähnlichen Effekten erbringen. Die Menge aller von einem Dienst d tatsächlich erbringbaren Effekte soll mit $\mathcal{D}$ bezeichnet werden. Es sei zudem s die Dienstbeschreibung dieses Dienst mit den IN_x-Variablen IN^1, IN^2, ..., IN^n. Der Raum aller nach s gültigen IN-Variablenbelegungen sei $\mathcal{I}$, ein Element hieraus wird mit (i_1, i_2, ..., i_n) bezeichnet.

Typischerweise kann der Dienst d bei einer gegebenen IN-Variablenbelegung eine Reihe möglicher Effekte erbringen, die konform zu s sind. Die *Effektmöglichkeiten* von d können daher durch eine Funktion ep_d (ep = effect possibilities) beschrieben werden:

$$ep_d : \mathcal{I} \longrightarrow \mathcal{P}(\mathcal{D}) \tag{1}$$

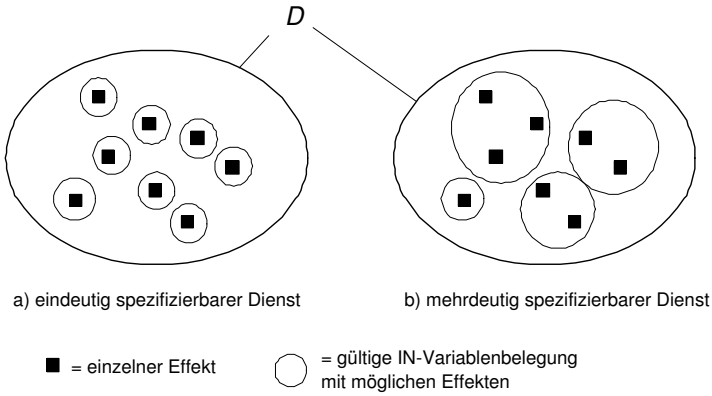


Abbildung 43: Unterscheidung zwischen (a) eindeutig und (b) mehrdeutig spezifizierbaren Diensten. Bei eindeutig spezifizierbaren Dienst legt jede IN-Variablenbelegung den zu erwartenden Effekt des Dienstes bereits eindeutig fest, bei mehrdeutig spezifizierbaren ist dies nicht der Fall. Hier wählt der Dienstgeber selbstständig einen Effekt aus der Menge der Effektmöglichkeiten aus.

Die Kardinalitäten der jeweiligen Menge der Effektmöglichkeiten bestimmen dann, ob ein Dienst eindeutig oder mehrdeutig spezifiziert ist (siehe dazu Abbildung 43):

- Ein Dienst ist genau dann *eindeutig spezifizierbar*, wenn für alle $i \in \mathcal{I}$ gilt: $|ep_d(i)| \leq 1$. Bei so spezifizierten Diensten ist also der zu erwartende Effekt nach Füllung aller IN-Variablen eindeutig festgelegt, sofern der Dienst mit dieser Variablenbelegung überhaupt ausgeführt werden kann (siehe dazu unterbestimmte Dienstbeschreibungen im nächsten Abschnitt). Beispiel für einen solchen Dienst ist ein Druckdienst, bei dem der Dienstnehmer über IN-Variablen alle Eigenschaften des Ausdrucks festlegen kann. Der zu erwartende Effekt ist also bekannt.
- Ein Dienst ist genau dann *mehrdeutig spezifizierbar*, wenn ein $i \in \mathcal{I}$ existiert, so dass gilt: $|ep_d(i)| > 1$. Bei so spezifizierten Diensten liegt der tatsächlich zu erbringende Effekt teilweise im Ermessen des Dienstgebers, in dem er selbstständig einen der möglichen Effekte aus der Menge $ep_d(i)$ der Effektmöglichkeiten auswählt. Der Dienstnehmer kann daher nur begrenzt Einfluss auf die Wirkung des Dienstes nehmen. Beispiel für einen solchen Dienst ist ein Dienst zur Reservierung von Kinokarten, bei dem der Dienstnehmer über IN-Variablen nur den gewünschten Film und eine Startzeit angeben kann. Die genaue Platz- und Saalnummer entscheidet der Dienstgeber jedoch eigenständig.

8.1.3 Unterbestimmte Dienstbeschreibung (*)

Bestimmte Dienste bieten an, einen Effekt für eine große Menge von Entitäten zu erbringen. Ein Beispiel hierfür könnte ein Dienst sein, über den ein Buch gekauft werden kann. Bei solchen Diensten ist es oft der Fall, dass die Entität, auf der der Dienst operiert, nicht zu einer öffentlichen Klasse gehört. Dies kann mehrere Gründe haben: Zum einen könnte es zu aufwändig sein, die Vielzahl der Instanzen allgemein zu veröffentlichen (z.B. alle existierenden Bücher), zum anderen könnte sich die Menge der Instanzen häufig ändern. Auch Situationen, in denen Informationen über Instanzen selbst einen Wert für den Dienstanbieter darstellen, können dafür ausschlaggebend sein, Klassen nicht als öffentliche Entitätsklassen zu definieren.

Abbildung 44 zeigt einen Ausschnitt aus dem Schema und einer beispielhaften Beschreibung eine Dienstes zum Kauf eines Buches. Die Auswahl des Buches (als teilöffentliche Entitätsklasse) findet

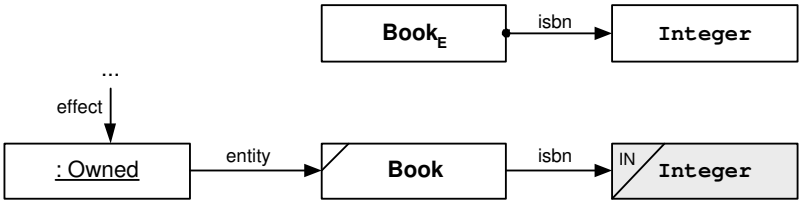


Abbildung 44: Ausschnitt aus einer Beschreibung für einen Dienst zum Verkauf von Büchern. Die Beschreibung kann aus zwei Gründen unterbestimmt sein: (a) nicht alle gültigen Integerwerte stellen ISBN-Nummern von existierenden Büchern dar, (b) der Dienst muss nicht alle existierenden Bücher verkaufen können.

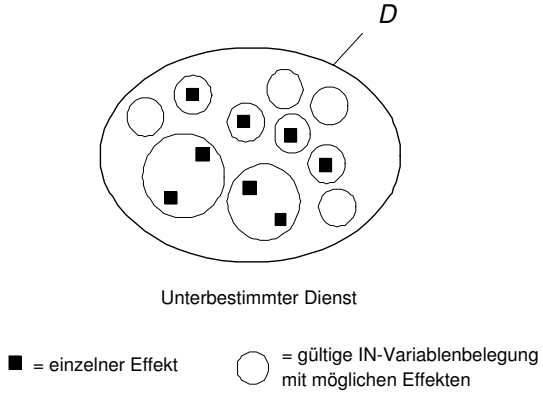


Abbildung 45: Eine Dienstbeschreibung ist unterbestimmt, wenn eigentlich gültige Belegungen der IN-Variablen zu leeren Mengen von Effektmöglichkeiten führen können.

dabei nicht über Angabe der Book-Instanz statt, sondern erfolgt eindeutig über das definierende Attribut isbn[12]. Dennoch kann es aus zwei Gründen dazu kommen, dass der Dienst nicht ausgeführt werden kann:

- Der Dienstnehmer wählt zwar typgerecht einen korrekten Füllwert für die IN-Variablen aus, dieser führt aber zu keiner allgemein veröffentlichen oder privat hinterlegten Instanz. Im Beispiel tritt dieser Fall ein, wenn der Dienstnehmer einen Integerwert angibt, der keiner ISBN-Nummer eines Buches entspricht, z.B. 18. In diesem Fall ist die Book-Menge leer; der Dienst kann nicht ausgeführt werden.
- Der Dienstnehmer wählt die Füllwerte für die IN-Variablen so, dass hierdurch eine nicht-leere Menge von Entitäten entsteht. Dies kann möglich sein, wenn er Werte von allgemein veröffentlichten oder Instanzen aus seinem privaten Instanzenpool beziehen kann. Im Beispiel könnte der Dienstnehmer eine ISBN-Nummer eines Buches angeben, dessen Instanz im allgemeinen Instanzenpool hinterlegt ist. Trotzdem könnte der Dienst die Ausführung verweigern, da er dieses Buch nicht liefern kann.

In beiden Fällen führt also eine eigentlich gültige Belegung der IN-Variablen zu leeren Mengen von Effektmöglichkeiten für den Dienstgeber, d.h. der Dienst kann nicht ausgeführt werden. Wir definieren wie folgt (siehe Abbildung 45):

[12] Es handelt sich daher um einen eindeutig spezifizierbaren Dienst.

- Eine Dienstbeschreibung ist genau dann *unterbestimmt*, wenn ein $i \in \mathcal{I}$ existiert, so dass $ep_d(i) = \emptyset$. Dienste, die mit einer solchen Variablenbelegung i aufgerufen werden, weisen daher den Aufruf vor der eigentlichen Ausführung ab.

Beim Vergleich von Dienstbeschreibungen muss der Vergleicher daher gegebenenfalls zusätzlich bestimmen, mit welcher Wahrscheinlichkeit ein Dienst mit einer bestimmten IN-Variablenbelegung auch ausgeführt werden kann. Diese *Ausführungswahrscheinlichkeit* ist ein Maß dafür, wie hoch die Chance ist, einen bestimmten Effekt tatsächlich erwirkt zu bekommen. Bei mehreren gleich gut passenden Dienstangeboten sollte daher der Dienst mit der höchsten Ausführungswahrscheinlichkeit bevorzugt werden. Weiterhin sollten Dienste mit sehr geringer Ausführungswahrscheinlichkeit nicht weiter betrachtet werden.

Der Vergleicher kann die Ausführungswahrscheinlichkeit nicht definitiv berechnen, sondern nur grob abschätzen. Hierbei können ihm zusätzliche Angaben in der Angebotsbeschreibung behilflich sein. Der Dienstgeber sollte bei jeder Menge vom Typ einer teilöffentlichen Klasse, deren Elemente über Pfade von Attributbedingungen durch IN-Variablen spezifiziert werden können, angeben, wieviele und welche Instanzen der Dienst unterstützt. Es existieren folgende Kardinalitätsmarkierer:

- `all values`. Gibt an, dass jede beliebige Belegung der IN-Variablen vom Dienst verarbeitet werden kann. Dieser Markierer kann vom Dienstgeber verwendet werden, wenn die IN-Variablen durch spezielle Typen oder genauere Bedingungen[13] in ihrer Grundmenge so eingegrenzt werden, dass alle Belegungen zu vom Dienst verarbeitbaren Entitäten führen.
- `all entities`. Gibt an, dass der Dienst immer ausgeführt werden kann, sofern die Belegung der IN-Variablen zu einer nicht-leeren Menge von Entitäten führt. Dabei können die enthaltenen Instanzen allgemein veröffentlicht oder im privaten Instanzenpool von Dienstnehmer oder -geber zu finden sein. Führt die Belegung zu einer leeren Menge von Entitäten, kann der Dienst nicht ausgeführt werden. Dies ist der Standardfall, falls kein Markierer spezifiziert ist.
- n `entities`. Gibt an, dass der Dienst auch dann nicht immer ausgeführt werden kann, wenn die Belegung der IN-Variablen zu einer nicht-leeren Menge von Entitäten führt, etwa weil der Dienst die spezifizierten Entitäten entgegen der Dienstbeschreibung nicht unterstützt. Die Zahl n gilt dann als Übersicht für die Dienstnehmer, wieviele Entitäten der Dienstgeber insgesamt unterstützt.

In g-dsd erfolgt die Angabe des Kardinalitätsmarkierers in geschweiften Klammern in einem eigenen Abschnitt der Mengendefinition. Die Beschreibung des Beispiels von oben ist daher erst mit der Angabe des Markierers wie in Abbildung 46 korrekt.

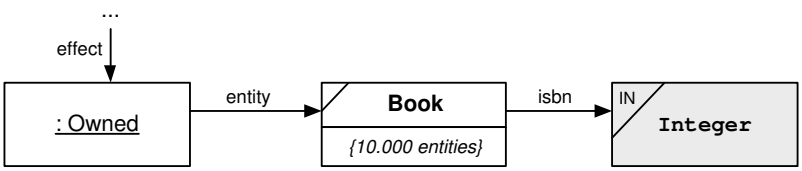


Abbildung 46: Beschreibung des Dienst zum Verkauf von Büchern mit Markierer n `entities`.

Der Markierer wird in f-dsd direkt bei der Mengendefinition nach der Typangabe in geschweiften Klammen notiert[14]. Das Beispiel von oben sieht also in f-dsd wie folgt aus:

[13] Häufig sind hierfür jedoch auch mengenübergreifende Bedingungen nötig, die in DSD zur Zeit noch nicht spezifiziert sind.

[14] Zur Zeit unterstützt der Parser die Angabe dieser Markierer noch nicht.

```
anonymous Owned
[
  entity =
  (
    set of Book {10000 entities}
    where isbn == var (in,x,1) from Integer
  )
]
```

In j-dsd ist die Angabe der Markierer noch nicht spezifiziert.

8.2 Anfragebeschreibungen

8.2.1 Überblick

Folgende Aufstellung gibt einen Überblick über die Besonderheit von Anfragebeschreibungen.

- *Fuzzymengen möglich.* Dienstnehmer haben die Möglichkeit, in der Anfrage detailliert zu spezifizieren, welche der geeigneten Effekte sie präferieren. Diese Präferenzen werden durch eine unscharfe Zugehörigkeitsfunktion zur Menge der Effekte ausgedrückt.
- *Grounding verweist auf gewünschten Dienstzugang.* Im Gegensatz zu Angebotsbeschreibungen stellt das Grounding einer Dienstanfrage eine Verbindung zwischen der Beschreibung der gewünschten Funktionalität und dem gewünschten Zugang zum Dienst dar. Dabei ist zu beachten, dass das Grounding nicht Teil der eigentlichen Anfrage ist und auch nicht für den Vergleich herangezogen wird. Vielmehr wird der gewünschte Zugangsweg *in jedem Fall* bereitgestellt, unabhängig vom tatsächlich aufgerufenen Dienst. Details hierzu finden sich in Abschnitt 10.
- *Keine Vorbedingungen.* Da der Anfrager in der Regel nicht weiß, welche Vorbedingung ein konkretes Dienstangebot benötigen wird, enthält eine Anfragebeschreibung typischerweise keine Beschreibung der vom Dienstnehmer erfüllbaren Vorbedingungen. Die Prüfung oder Erfüllung der Vorbedingungen einer Angebotsbeschreibung finden erst dann statt, wenn für die Effekte bereits ein positives Vergleichsergebnis festgestellt wurde. Details hierzu finden sich in Abschnitt 9.

9 Vorbedingungen

Dieses Kapitel beschreibt Vorbedingungen von Diensten. Zunächst wird der Begriff geklärt und untersucht, wo Vorbedingungen zum Einsatz kommen. Anschließend wird vorgestellt, wie Vorbedingungen in DSD verwendet werden und ein Beispiel dafür gegeben. Abschließend wird die Beeinflussbarkeit von Vorbedingungen, ihre Auswertbarkeit und ihr Zusammenhang zur Dienstkombination erläutert.

9.1 Begriff und Einsatz von Vorbedingungen

Dieses Kapitel beschreibt, wie Vorbedingungen in Dienstbeschreibungen integriert werden können. Unter einer *Vorbedingung* (engl. precondition) eines angebotenen Dienstes versteht man eine Bedingung, die erfüllt sein muss, damit der Dienst erfolgreich ausgeführt werden kann. Ein Dienst kann keine, eine oder mehrere Vorbedingungen besitzen. Ist eine der Vorbedingungen beim Anstoßen der Dienstausführung nicht erfüllt, ist das Ergebnis der Dienstausführung nicht definiert: Entweder wird die Dienstausführung direkt vom Dienstgeber abgelehnt, ohne dass es zu einer Durchführung des Dienstes kommt, oder die Dienstausführung wird gestartet, aber fehlerhaft oder unvollständig beendet.

Vorbedingungen kommen nur in Beschreibungen von angebotenen Diensten vor. Prinzipiell könnte auch in Anfragebeschreibungen vom Dienstnehmer notiert werden, welche Vorbedingungen erfüllt werden können, dies macht jedoch wenig Sinn, da im Voraus oft nicht bekannt ist, welche konkreten Vorbedingungen für ein Dienstangebot erfüllt sein müssen. In Beschreibungen von benötigten Diensten existieren daher keine Vorbedingungen. Vielmehr unterhält der Dienstnehmer einen *privaten Instanzenpool*, in dem er persönliche benamte Instanzen von teilöffentlichen Entitätsklassen ablegt, die nicht für die Allgemeinheit zugänglich sein sollen, wie etwa Angaben zu seiner Person, Kreditkarten- oder Kontoinformationen, Informationen zu seinem Aufenthaltsort, zu seinem Rechner etc. Allgemein wird hier also der Zustand festgehalten, in dem sich der Benutzer und seine Umgebung aktuell befinden. Diese Informationen können dann vom Vergleicher verwendet werden, um zu überprüfen, ob die vom Dienstgeber geforderten Vorbedingungen erfüllt sind.

9.2 Vorbedingungen in DSD

In DSD gilt folgender Grundsatz für das Einbringen von Vorbedingung in Dienstbeschreibungen: Vorbedingungen werden nicht in Form spezieller Formeln notiert und dann als weiterer Teil an die Beschreibung gehängt, sondern es existieren allgemein vereinbarte Bedingungen zu jeder Angebotsbeschreibung, die genau dann erfüllt sein sollen, wenn die Vorbedingung erfüllt sind. In DSD werden zwei Arten von Vorbedingungen unterschieden:

- *Informationsvorbedingungen* sind Bedingungen, die sicherstellen, dass der Dienstgeber mit allen Informationen versorgt wird, die er zum korrekten Ausführen des Dienstes benötigt.
- *Zustandsvorbedingungen* sind Bedingungen, die sicherstellen, dass beim Start der Dienstausführung bestimmte Eigenschaften der Welt gegeben sind, die zum korrekten Ablauf des des Dienstes erforderlich sind.

Diese können auf folgende generische Bedingungen abgebildet werden, so dass die Vorbedingungen des Dienstes genau dann erfüllt sind, wenn diese beiden vereinbarten Regeln erfüllt sind. Zur

Überprüfung der Vorbedingungen werden daher diese Regeln überprüft, was meist rein syntaktisch geschehen kann:

Generische Vorbedingung 1: Eindeutige IN-Variablen-Belegung
Jede OffIN_x-Variable muss vor Beginn der Ausführung des Dienstes korrekt und eindeutig gebunden sein.

Durch Angabe von IN_x-Variablen kann der Dienstgeber somit seine Informationsvorbedingungen ausdrücken. Ist eine OffIN_x-Variable bei der Anstoßung der Ausführung nicht eindeutig[15] gebunden, kann der Dienst nicht ausgeführt werden.

Generische Vorbedingung 2: Keine leere Mengen
Keine der in der Angebotsbeschreibung angegebenen Mengen darf zu Beginn der Dienstausführung (d.h. nach der Bindung der OffIN_x-Variablen) leer sein.

Durch die Sicherstellung, dass keine der angegebenen Mengen leer sein darf, kann der Dienstgeber seine Zustandsvorbedingungen ausdrücken. Für eine Zustandsvorbedingung ausgedrückt durch die Menge m existieren drei Möglichkeiten:

- Kommt die Menge m als Bestandteil einer Effektbeschreibung vor, gilt diese automatisch als Zustandsvorbedingung und darf nicht leer werden.
- Kommt die Menge m nicht als Bestandteil einer Effektbeschreibung vor und ist vom Typ State oder einem Untertyp, so wird m direkt als ein Füllwert des listenwertigen Attributs precondition von ServiceProfile eingesetzt.
- Kommt die Menge m nicht als Bestandteil einer Effektbeschreibung vor und ist nicht vom Typ State oder einem Untertyp, so wird als ein Füllwert des Attributs precondition eine anonyme Instanz des Zustands Exists aus `upper.profile` eingetragen (siehe Abbildung 47), welche m als Füllwert für ihr Attribut entity hat.

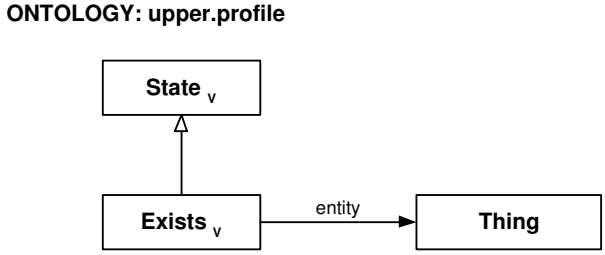


Abbildung 47: Zustand Exists aus der Ontologie `upper.profile`

9.3 Beispielbeschreibung mit Vorbedingungen

Abbildung 48 zeigt ein Beispiel, das die Verwendung von Vorbedingungen in DSD illustriert. Beschrieben ist ein E-Commerce-Dienst, bei dem ein Buch gekauft werden kann. Sein Profil besteht

[15] In Zukunft soll diese Bedingung verfeinert werden: Eine Variable muss dann gemäß der angegebenen *Bindungsmöglichkeiten* gebunden sein, was auch eine enumerierte, deklarative oder leere Bindung zulassen kann.

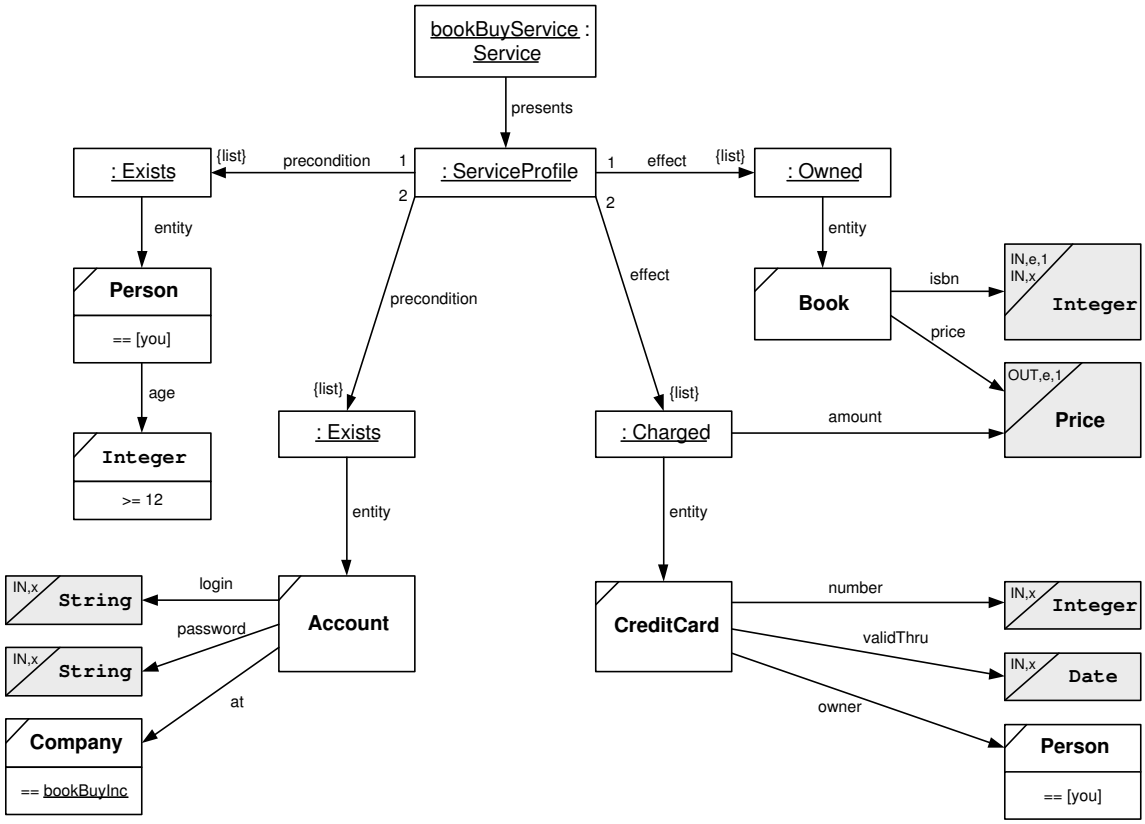


Abbildung 48: Beispielhafte Angebotsbeschreibung, die die Verwendung von Vorbedingungen in DSD illustriert.

aus zwei gefüllten effect-Attributen: Der erste Effekt drückt aus, dass sich das ausgewählte Buch nach der Dienstausführung im Besitz des Dienstnehmers befinden werden kann. Die Auswahl erfolgt über die ISBN-Nummer des Buches, die über eine IN_x-Variable angegeben wird. Da dieser Haupteffekt den Zustand eines realweltlichen Objekts verändert, handelt es sich bei dem Dienst um einen Realweltdienst. Der zweite Effekt ist die Belastung einer Kreditkarte, welche über die Nummer und ihr Gültigkeitsdatum ausgewählt wird. Da amount von Charged und price von Book die gleiche Preismenge als Attributbedingung haben, wird die Kreditkarte um den Betrag belastet, der das Buch kostet. Der Dienst bietet zudem die Möglichkeit, eine vorgelagerte Schätzphase zu nutzen. Ausgedrückt wird dies durch die $\mathsf{IN}_{e,1}$- und $\mathsf{OUT}_{e,1}$-Variablen an Book. Nach Angabe der ISBN-Nummer liefert der Dienst den Preis des Buches zurück, ohne dieses zu kaufen.

Der Beispieldienst enthält verschiedene Arten von Vorbedingungen. Zunächst gibt es Informationsvorbedingungen, die durch die verwendeten IN_x-Variablen ausgedrückt werden. Damit dieser Dienst ordnungsgemäß ablaufen kann, muss der Dienstgeber die ISBN-Nummer des gewünschten Buchs, die Nummer und das Gültigkeitsdatum seiner Kreditkarte, sowie Login und Passwort seines Accounts angeben. Fehlt eine dieser Informationen, wird der Dienstgeber die Ausführung zurückweisen.

Weitere Vorbedingungen stellen die Zustandsvorbedingungen dar, die durch jede verwendete Menge repräsentiert werden. Diese dürfen zu Beginn der Dienstausführung nicht leer sein. Im Beispiel sind das die folgenden Mengen:

- Die Person-Menge. Diese enthält nur Personen, die einerseits gleich dem Dienstnehmer (==

`[you]`[16]) sind, anderseits nur Personen, die 12 Jahre oder älter sind. Da die Menge nicht leer sein darf, wird hierdurch ausgedrückt, dass der Dienstnehmer mindestens 12 Jahre alt sein muss, um den Dienst nutzen zu können. Hierbei ist jedoch zu beachten, dass die Bedingung nur vom Dienstnehmer selbst überprüft werden kann, da er die Füllwerte der eigenen Person-Instanz ausschließlich in seinem privaten Instanzenpool hält. Die ist möglich, da Person ist eine teilöffentliche Entitätsklasse ist. Der Dienstgeber hingegen kann die Bedingung nicht testen, da er die Instanz nicht kennt und beim Aufruf des Dienstes keine weiteren Informationen darüber erhält.

- Die Account-Menge. Diese enthält nur Accounts mit dem angegebenen Login und Passwort, die bei der Firma `bookBuyInc` registriert sind. Da Account eine teilöffentliche Entitätsklasse ist, sind die Instanzen, die die Accounts repräsentieren, nicht allgemein veröffentlicht, sondern liegen lokal beim Dienstgeber vor. Der Dienstnehmer muss daher die korrekten Werte für die Variablenbindung kennen, da die Menge sonst leer und die Ausführung vom Dienstgeber zurückgewiesen wird.
- Die CreditCard-Menge. Diese verhält sich ähnlich wie die Account-Menge. Auch sie enthält nur Kreditkarten-Instanzen mit einer durch IN-Variablen bestimmten Nummer und Gültigkeitsdatum. Zudem müssen sie dem Dienstnehmer gehören. Auch CreditCard ist eine teilöffentliche Entitätsklasse, aber im Gegensatz zu Account liegen die Instanzen in den privaten Instanzenpools der Dienstnehmer. Diese können verwendet werden, um die geforderten IN-Variablen korrekt und eindeutig zu füllen. Der Dienstgeber auf der anderen Seite muss sich anderer Tests bedienen, um zu überprüfen, ob die Menge nicht leer ist, d.h. ob die angegebene Kreditkarte existiert und im Besitz des Dienstnehmers ist.
- Die Book-Menge. Diese enthält nur Bücher mit der angegebenen ISBN-Nummer. Sie kann leer werden, wenn der Dienstnehmer eine ISBN-Nummer spezifiziert, die keinem Buch entspricht. In diesem Fall wird die Dienstausführung vom Dienstgeber zurückgewiesen. Da ein Kardinalitätsmarkierer fehlt (siehe Abschnitt 8.1.3), gilt `all entities`, d.h. der Dienst kann jedes Buch, für das eine Instanz existiert, liefern.

Das Beispiel zeigt, dass der Dienst insgesamt fünf Informationsvorbedingungen und vier Zustandsvorbedingungen enthält, das listenwertige Attribut precondition jedoch nur mit zwei Füllwerten gefüllt ist. Dies liegt daran, dass jede IN_x-Variable und jede Menge eine Vorbedingung darstellt. Das Attribut precondition dient nur dazu, Mengen aufzunehmen, die nicht im Rahmen eines Effekts auftreten (wie hier Person[17] und Account). Im Folgenden versteht man daher unter Vorbedingungen nicht die Füllwerte des precondition-Attributs, sondern alle Informations- und Zustandsvorbedingungen.

9.4 Beeinflussbarkeit und Auswertbarkeit von Vorbedingungen

Bei Vorbedingungen wird unterschieden, ob diese vom Dienstnehmer beeinflussbar sind oder nicht, d.h. es man untersucht, ob es für den Dienstnehmer möglich ist, unerfüllte Vorbedingungen gezielt zu erfüllen. Man definiert daher:

- Eine Vorbedingung heißt *beeinflussbar*, wenn der Dienstnehmer prinzipiell die Möglichkeit hat, sie gezielt zu erfüllen. Typischerweise sind alle Informationsvorbedingungen beeinflussbar. Im Beispiel aus Abbildung 48 sind zudem die Zustandsvorbedingungen zum Account, Book und zur CreditCard beeinflussbar, da der Dienstnehmer hier dir Möglichkeit hat, diese durch geeignete Bindungen der anhängenden IN_x-Variablen zu erfüllen.

[16] `[you]` bezeichnet immer die Instanz von Person, die den Dienstnehmer repräsentiert.

[17] Die Bedingung, dass der Dienstnehmer mindestens 12 Jahre alt ist, hätte auch an die Person-Menge vom Kreditkartenbesitzer angehängt werden können, was einen weiteren precondition-Füllwert gespart hätte.

- Eine Vorbedingung heißt *nicht beeinflussbar*, wenn der Dienstnehmer keinen Einfluss darauf hat, ob diese zu Beginn der Dienstausführung erfüllt ist oder nicht. Im Beispiel ist dies die Zustandsvorbedingung zu Person.

Zudem werden Zustandsvorbedingungen dahingehend unterschieden, ob und von wem sie ausgewertet werden können:

- Eine Zustandsvorbedingung heißt *vom Dienstgeber direkt auswertbar*, falls dieser in der Lage ist, die Leere oder Nichtleere der zugehörigen Menge allein aufgrund seines ontologischen Wissens (bestehend aus allgemeinem Wissen über Schema und öffentliche Instanzen und Wissen über persönliche Instanzen im privaten Instanzenpool) festzustellen. Im Beispiel sind das die Vorbedingung zu Book und Account.
- Eine Zustandsvorbedingung heißt *vom Dienstgeber indirekt auswertbar*, falls dieser in der Lage ist, die Leere oder Nichtleere der zugehörigen Menge mittels zusätzlicher, auch externer Wissensquellen und Dienste festzustellen. Diese Feststellung ist im Normalfall mit einer gewisser Fehlerrate behaftet. Im Beispiel kann die Zustandsvorbedingung zur CreditCard wie oben angedeutet vom Dienstgeber nur indirekt ausgewertet werden, indem er beispielsweise einen weiteren Dienst nutzt.
- Eine Zustandsvorbedingung heißt *vom Dienstnehmer auswertbar*, falls dieser in der Lage ist, lokal bestimmen zu können, ob die zugehörige Menge leer ist oder nicht, ohne zuvor die Dienstausführung anstoßen zu müssen. Auch hier kann zwischen direkter und indirekter Auswertbarkeit unterschieden werden. Im Beispiel sind das die Bedingungen zu Person und CreditCard. Die Bedingung zu Account kann zwar auch ausgewertet werden, jedoch kann der Account zwischenzeitlich gelöscht worden sein. Die Bedingung zu Book ist für den Dienstnehmer nicht auswertbar, da nicht bekannt ist, welche Buchinstanzen sich im privaten Instanzenpool des Dienstgebers befinden.

Die Auswertbarkeit hat Einfluss darauf, welche Ausführungs- und Fehlerwahrscheinlichkeit für den Dienst besteht. Ist eine Bedingung zwar vom Dienstgeber auswertbar, vom Dienstnehmer aber nicht, so kann es vorkommen, dass der Dienst so aufgerufen wird, dass diese Bedingung vor der Dienstausführung nicht erfüllt ist. Dies wird jedoch vom Dienstgeber erkannt, der die Ausführung daraufhin abweist. Für den Dienstnehmer hat der Dienst dann eine Ausführungswahrscheinlichkeit kleiner als 1.0. Im Beispiel könnte der Dienstnehmer beispielsweise eine ISBN-Nummer angeben, die zur einer leeren Buchmenge führt.

Ist eine Bedingung vom Dienstgeber nicht oder nur ungenau auswertbar, kann der Fall eintreten, dass die Dienstausführung vom Dienstgeber nicht zurückgewiesen wird, obwohl die Bedingung vor der Dienstausführung nicht erfüllt war. In diesem Fall läuft die Dienstausführung unter falschen Voraussetzungen, was zu fehlerhaften, unvollständigen oder unerwünschten Ergebnissen führen kann. Wenn sie nachträglich erkannt werden, können sie gegebenenfalls durch Kompensationsfunktionen rückgängig gemacht werden. Im Beispiel kann die Bedingung zu Person vom Dienstgeber nicht ausgewertet werden. Daher kann es zu dem unerwünschten Effekt kommen, dass ein Geschäft mit einer nicht geschäftsfähigen Person durchgeführt wird.

9.5 Vorbedingungen und Dienstkombination

Beim Vergleich von Dienstbeschreibungen auf Seiten des Dienstnehmers ist zu beachten, dass auswertbare und vom Dienstnehmer beeinflussbare Bedingungen, die für einen angeboten Dienst nicht erfüllt sind, nicht direkt zur Verwerfung dieses Diensten führen müssen. Es kann zunächst versucht werden, die Bedingung auf anderem Wege zu erfüllen, bevor die Dienstausführung angestoßen wird:

- Nicht eindeutig bindbare IN_x-Variablen können mit persönlichen Instanzen aus dem privaten Instanzenpool gefüllt werden.
- Fehlende Informationen zum Binden von IN_x-Variablen können durch Nutzung von Wissensdiensten beschafft werden.
- Bei Auftreten leerer Mengen kann versucht werden, ein geeignetes Element über einen anderen Dienst zu erzeugen. Hierzu können Informations-, Realwelt- oder Befähigungsdienste dienen.

Durch Nutzung von weiteren Diensten zur Erfüllung von Vorbedingung des eigentlichen Dienstes entsteht eine Kette von Diensten, die insgesamt den gewünschten Effekt erbringt. Durch diesen Ansatz ist es möglich, Dienste automatisch zur Laufzeit zu neuen Diensten zu kombinieren.

10 Definieren von Groundings

Ein Teil einer Dienstbeschreibung in DSD stellt das *Grounding* dar.[18] Das Grounding beschreibt den Zusammenhang zwischen der abstrakten, formellen Beschreibung des Dienstes im Profile und der konkreten, ausführbaren Funktion, welche als Dienst beschrieben ist. Im Gegensatz zu anderen Sprachen existiert in DSD sowohl für Angebots- als auch für eine Anfragebeschreibungen ein Grounding:

- Das Grounding eines Dienstangebots beschreibt den Zusammenhang zu einer *real existierenden* Funktion, die als Dienst bereitgestellt werden soll. Eine Funktion kann hierbei beispielsweise eine Javaklasse mit ihren Methoden, ein Web Service oder ein Wrapper zu einer Webseite sein.
- Das Grounding einer Dienstanfrage beschreibt den Zusammenhang zu einer *gewünschten* Funktion, die beispielsweise innerhalb einer Applikation verwendet und über einen dynamischen Dienstaufruf realisiert werden soll.

Das Grounding hat in beiden Fällen zwei Aufgaben. Zum einen beschreibt es den Zusammenhang des Informationsflusses, d.h. es legt fest, wie der durch die IN- und OUT-Variablen spezifizierte Nachrichtenfluss in der zugrundeliegenden Funktion ausgeführt werden muss. Konkret wird also angegeben, wie die Werte der IN-Variablen als Parameter an die Funktion übergeben werden und wie die Daten der OUT-Variablen als Rückgabewerte der Funktion abgelesen werden. Zum zweiten beschreibt das Grounding den Zusammenhang zum Zustandsübergang, d.h. es gibt an, wie die Funktion konkret verwendet muss, damit die im Profil angegebenen Effekte tatsächlich entstehen.

10.1 Schema des Groundings in DSD

Das Schema des Groundings wurde in [2] entwickelt und ist in Abbildung 49 dargestellt. Ausgangspunkt ist die Klasse ServiceGrounding, die von der Klasse Service in `upper` über das Attribut supports eingebunden werden kann. ServiceGrounding muss als abstrakte Klasse angesehen werden,

[18] Der Begriff wird in ähnlicher Weise in OWL-S verwendet und ist dort entliehen.

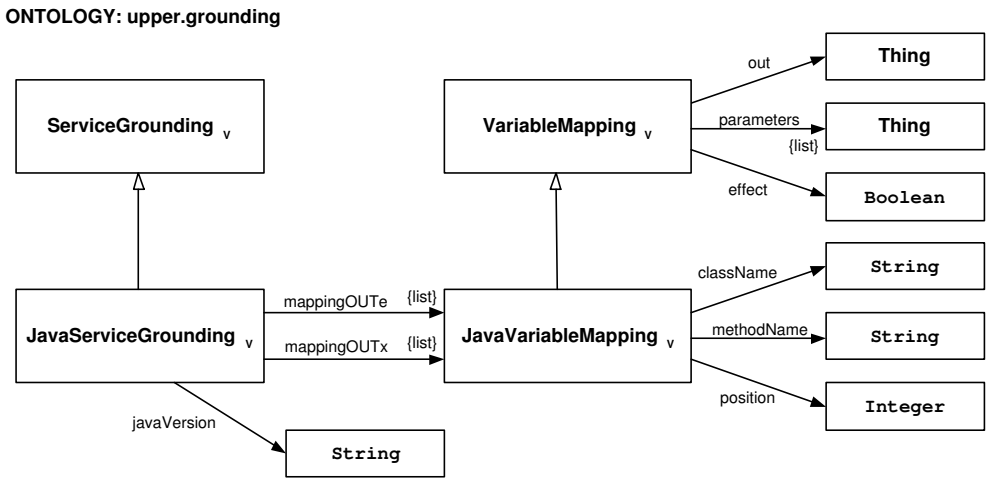


Abbildung 49: Ontologie `upper.grounding` für Groundings in DSD.

denn es existieren keine direkten Instanzen hiervon. Vielmehr stehen für die verschiedenen technischen Umsetzungen der eingebundenen Funktionen unterschiedliche spezialisierte Groundingklassen zur Verfügung. In Abbildung 49 ist dies beispielhaft für Groundings auf Javamethoden angegeben. Grounding auf andere Programmierumgebungen sind in Arbeit.

Das JavaServiceGrounding beschreibt zunächst mittels des Attributs javaVersion welche Version vorhanden sein muss, damit die zugrundeliegende Funktion verwendet werden kann. Wichtiger sind jedoch die listenwertigen Attribute mappingOUTe und mappingOUTx. Für jede in der Dienstbeschreibung auftretende OUT-Variable muss die Klasse JavaVariableMapping einmal instanziiert und in das entsprechende listenwertige Attribut eingefüllt werden. Weitere Instanzen sind für Methoden nötig, die keine Ausgaben in Form von Rückgabewerte besitzen. JavaVariableMapping beschreibt, wie der Wert einer OUT-Variablen durch die reale Funktion berechnet wird. Dazu verweist das geerbte Attribut out auf die entsprechenden OUT-Variable im Profile. className und methodName geben an, welche Javamethode aufgerufen wird. Als Parameter werden ihr die in parameters angegebenen Werte von IN-Variablen übergeben. Mithilfe des Attributs effect wird festgelegt, ob der Aufruf dieser Methode die im Profile beschriebenen Effekte erbringt. Dieses ist nur für Mappings sinnvoll, die in mappingOUTx enthalten sind. Existieren in der Javaklasse Methoden ohne Rückgabewert, die jedoch einen der beschriebenen Effekte erbringen, so werden hierfür dennoch JavaVariableMappings instanziiert, ihr out-Attribut wird jedoch nicht gefüllt. Da Effekte nur in der Ausführungsphase auftreten können, werden sie in mappingOUTx eingefüllt.

Das Attribut position hat für Groundings von Angebots- und Anfragebeschreibungen eine unterschiedliche Bedeutung. In Angebotsbeschreibungen legt es fest, in welcher Reihenfolge die Methoden aufgerufen sollen, wenn darüber Unklarheit herrscht (etwa bei mehreren Mappings ohne OUT-Variable). In einer Anfragebeschreibungen legt das Attribut fest, in welcher Reihenfolge die Werte der OUT-Variablen im Rückgabetyp der gewünschten Funktion auftreten sollen.

10.2 Beispiel für ein Grounding

Für ein beispielhaftes Grounding einer Angebotsbeschreibung soll der Dienst aus Kapitel 9 wieder aufgegriffen werden (siehe Abbildung 48, Seite 71). Wir nehmen an, die Funktionalität dieses Verkaufsdiensts sei in Java programmiert und stehe über die folgende Klasse `BookShop` mit den Methoden `login`, `priceOfBook` und `buyBook` zur Verfügung:

```
import dsd.schema.domain.money.Price;

public class BookShop
{
   public static void login(String accountName, String passwd)
   {...}

   public static Price priceOfBook(int isbn)
   {...}

   public static void buyBook(int isbn, int ccnumber, GregorianCalendar ccvalid)
   {...}
}
```

Abbildung 50 zeigt das zugehörige Grounding auf die Javamethoden in g-dsd. Nötig sind drei Instanzen von JavaServiceGrounding: Zwei als Füllwerte von mappingOUTx, die die Abbildung auf die `login`- und `priceOfBook`-Methode vornehmen. Da die Methoden keine Rückgabewerte besitzen,

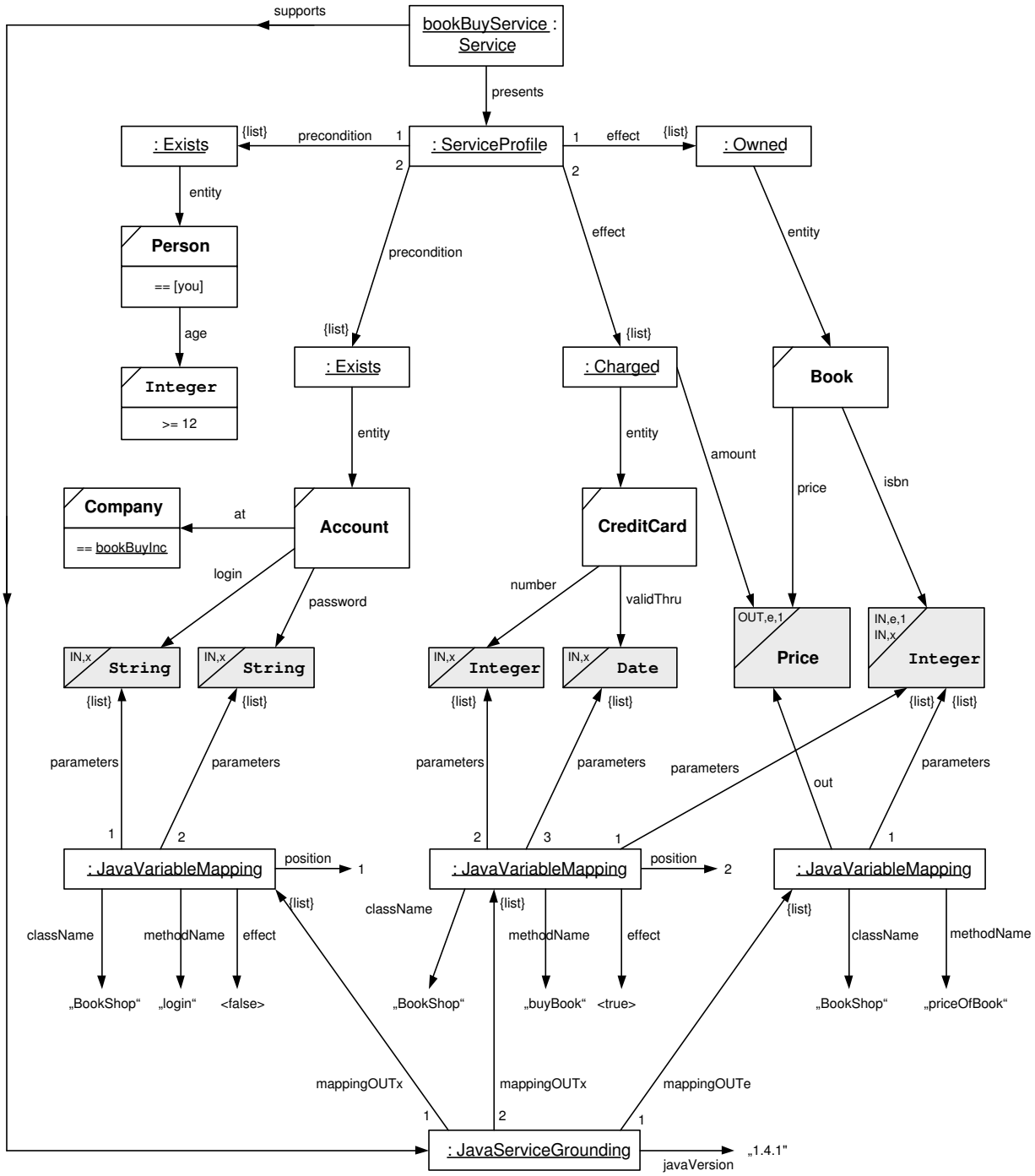


Abbildung 50: Beispielhafte Angebotsbeschreibung mit Grounding auf Javamethoden.

bleibt das Attribut out jeweils unausgefüllt. Nur die Methode buyBook erbringt den beschriebenen Effekt, nämlich dass ein Buch gekauft und die Kreditkarte belastet wird. Im Mapping ist daher das Attribut effect auf <true> gesetzt. Das mappingOUTe beschreibt die Hilfsmethode für die Schätzphase. Ausgabe ist hier der Wert der Price-Variable, was im Attribut out notiert ist.

10.3 Mapping zwischen Datentypen

Bei der Abbildung zwischen einer realen Funktion und DSD treffen Datentypen aus unterschiedlichen Umgebungen aufeinander: Typen aus DSD und Typen aus der Programmierumgebung der Funktion. Der Zusammenhang zwischen diesen wird nicht in jeder Dienstbeschreibung erneut angegeben, sondern ist allgemein für jede Unterklasse von ServiceGrounding festgelegt. Generell gilt, dass die primitiven Datentypen von DSD direkt auf eingebaute Datentypen der Programmierumgebung der beschriebenen Funktion abgebildet werden sollten. Jedoch sind auch Abbildungen auf selbstdefinierte Typen denkbar. Klassentypen aus DSD können im Normalfall nicht auf das Typsystem der Programmierumgebung abgebildet werden.

Für JavaServiceGroundings gilt die folgende besondere Regelung zur Typabbildung:

- Primitive Datentypen von DSD werden direkt auf korrespondierende primitive Typen oder Javaklassen abgebildet. Die genaue Umsetzung liefert die Tabelle in Abbildung 51.
- Auch DSD-Klassen können in Java verwendet werden. Sie werden dazu auf die entsprechenden Javaklassen der j-dsd-Repräsentierung abgebildet. Die Java-Methoden der beschriebenen Funktionen müssen in solchen Fällen direkt mit den j-dsd-Typen arbeiten. Im Beispiel von oben verwendet beispielsweise die Javamethode `priceOfBook` als Rückgabewert den DSD-Typ `dsd.schema.domain.money.Price`. Eine weiter gehende Abbildung von dsd-Klassen auf Javaklassen ist im Grounding nicht vorgesehen, sondern muss durch externe Wrapper durchgeführt werden.

Primitiver Typ in DSD	Typ in Java
Integer	int
Double	double
String	java.lang.String
Boolean	boolean
Date	java.util.GregorianCalendar
Time	java.util.GregorianCalendar
DateTime	java.util.GregorianCalendar
Duration	long

Abbildung 51: Umsetzung der primitiven Datentypen von DSD auf Typen in Java.

10.4 Besonderheiten in f-dsd

Bei der Notation eines Groundings in f-dsd ist zu beachten, dass die Beschreibung insgesamt keinen baumartigen Aufbau mehr besitzt, sondern insbesondere die Variablen mehrmals als Füllwerte dienen. Realisiert wird dies in f-dsd durch Benennen und Referenzieren von Mengen bzw. Variablen.

Hierzu wird bei ihrer ersten Definition ein *Label* vergeben. Bei jedem weiteren Verweis auf die Menge/Variable wird dann dieses Label verwendet. Labels in f-dsd sind kleingeschriebene Bezeichner, die mit einem `$` beginnen.

Die Definition eines Label erfolgt durch Angabe des Namens des Labels

- bei Variablen direkt nach dem Schlüsselwort `var`;
- bei Mengen direkt vor dem Schlüsselwort `set`;
- bei anonymen Instanzen direkt nach dem Schlüsselwort `anonymous`.

Die Referenzierung des durch das Label benannten Objekts erfolgt durch Angabe des Namens.

In f-dsd sieht daher ein Teil der Dienstbeschreibung aus Abbildung 50 wie folgt aus:

```
...
presents = anonymous ServiceProfile
[
  ...
  (
    set of Book
    where price == var $price (out,e,1) from Price
    and isbn == var $isbn (in,e,1) (in,x,1) from Integer
  )
  ...
],
supports = anonymous JavaServiceGrounding
[
   javaVersion = "1.4.1",
   mappingOUTe += anonymous JavaVariableMapping
   [
      out = $price,
      parameters += $isbn,
      className = "BookShop",
      methodName = "priceOfBook"
   ]
   ...
]
...
```

10.5 Groundings für Informationsdienste (*)

Eine Aufgabe des Groundings ist die Angabe, wie die reale Funktion aufgerufen werden muss, damit die im Profile angegebenen Effekte erbracht werden. Im Falle von Wissensdiensten erfolgt der Effekt durch das Bekanntwerden der Füllwerte für die OffOUT-Variablen und ist damit bereits durch das Grounding zum Informationsfluss beschrieben. Im Falle von Realweltdiensten erfolgt der Effekt in der realen Welt. Neben dem Verweis auf die auslösende Funktion und dem Setzen des Attributs effect auf <`true`> ist daher keine weitere Beschreibung möglich.

Kritischer sind Informationsdienste. Diese erbringen ihren Effekt innerhalb des Informationssystems, meist durch Austausch von Dateien oder Datenbankeinträgen. Hier sollten im Grounding die technischen Details zu diesem Vorgang vermerkt werden, etwa welches Austauschprotokoll verwendet wird und wie dieses genau zu konfigurieren ist. Geplant ist eine Änderung der betroffenen Datentypen File und DatabaseEntry. Hier soll das Attribut bytes nicht mehr den generischen Typ `String` besitzen, sondern durch einen neuen Typ ByteStream ersetzt werden. Für Variablen dieses Typs könnte dann im Grounding das verwendete Austauschprotokoll beschrieben werden.

11 Zusammenfassung und Ausblick

In diesem Handbuch wurde die semantische Dienstbeschreibungssprache *DIANE Service Description* (DSD) vorgestellt. Zunächst wurden die neuartigen Konzepte präsentiert, dann erläutert, wie mit DSD Ontologien bestehend aus Schemata, Instanzen, Mengen und Variablen beschrieben werden können und abschließend gezeigt, wie Dienstanfrage- und -angebotsbeschreibungen aufgestellt werden können. Als Forschungsgegenstand unterliegt DSD ständigen Änderungen und Erweiterungen. Daher ist die neuste Version dieses Dokuments stets auch online verfügbar[19].

Eine Reihe von Themen um DSD sind in diesem Handbuch nicht angesprochen worden. Hierzu zählen:

- Der Vergleich von Dienstbeschreibungen. Details hierzu finden sich in [14] und [16].
- Die dienstorientierte Middleware auf Java-Basis, die ebenfalls im DIANE-Projekt entwickelt wurde und es erlaubt, mit DSD beschriebene Dienste automatisch nutzen zu können. Sie besteht aus einer Reihe von Komponenten (auch Agenten genannt) auf Seiten des Dienstnehmers und einem Containermanager (siehe [2]) auf Seiten des Dienstgebers.
- Iteratoren in Dienstbeschreibungen. Diese werden verwendet, wenn nicht ein Effekt des Dienstes von Bedeutung ist, sondern mehrere oder alle. Iteratoren wirken sich auf die Beschreibung, den Vergleich und die Ausführung von Diensten aus.
- Weitere sprachliche Erweiterungen wie kontextabhängige Instanzen (wie `[you]` oder `[here]`), deklarative Variablenbindung, optionale Variablenbindung, mengenübergreifende Bedingungen und vieles mehr.

Sie sollen evtl. in folgenden Versionen aufgenommen werden.

[19] unter `http://www.ipd.uka.de/DIANE/docs/DSD-Cookbook.pdf`

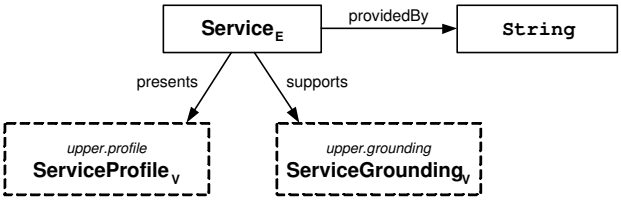
Teil IV

Anhang

A Wichtige Ontologien

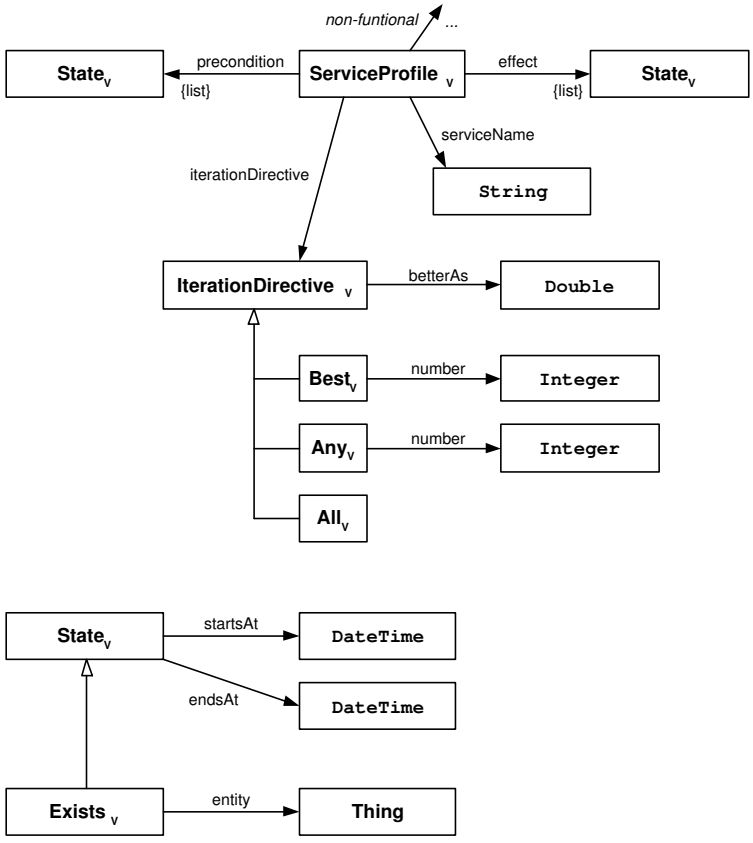
A.1 upper

ONTOLOGY: upper



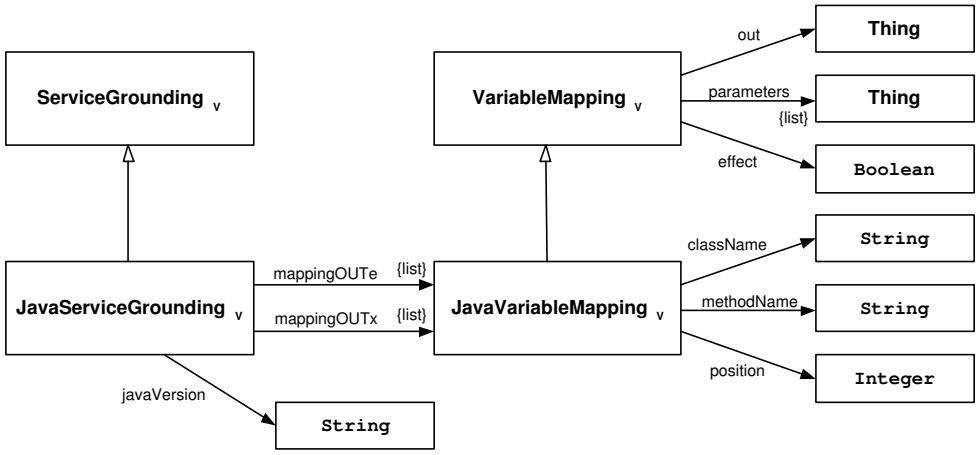
A.2 upper.profile

ONTOLOGY: upper.profile



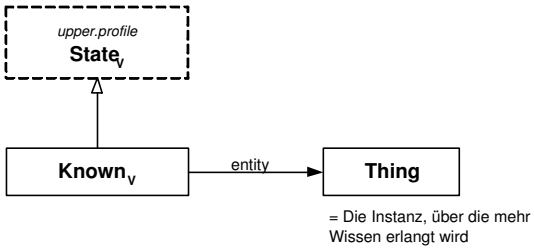
A.3 upper.grounding

ONTOLOGY: upper.grounding



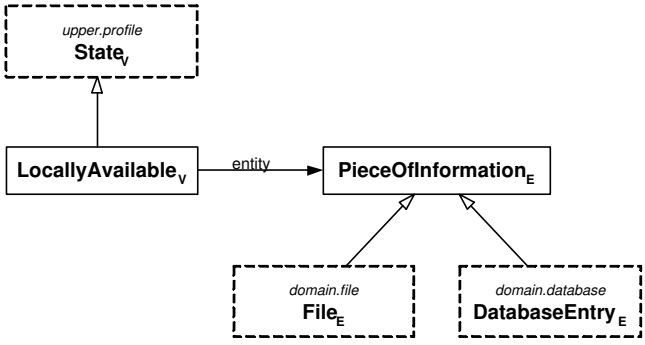
A.4 category.knowledgeservice

ONTOLOGY: category.knowledgeservice



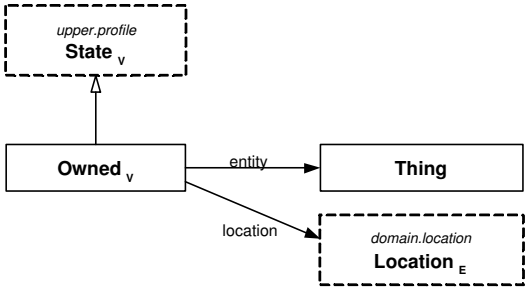
A.5 category.informationservice

ONTOLOGY: category.informationservice



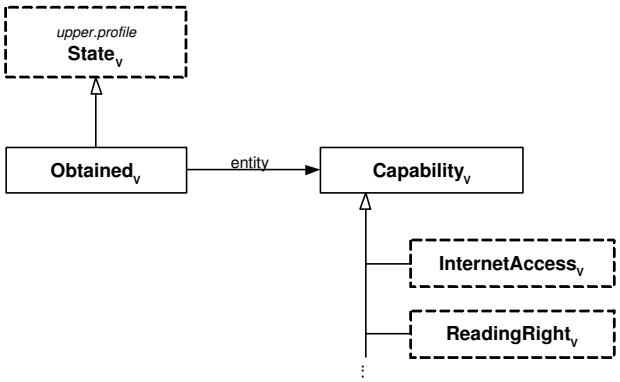
A.6 category.realobjectservice

ONTOLOGY: category.realobjectservice



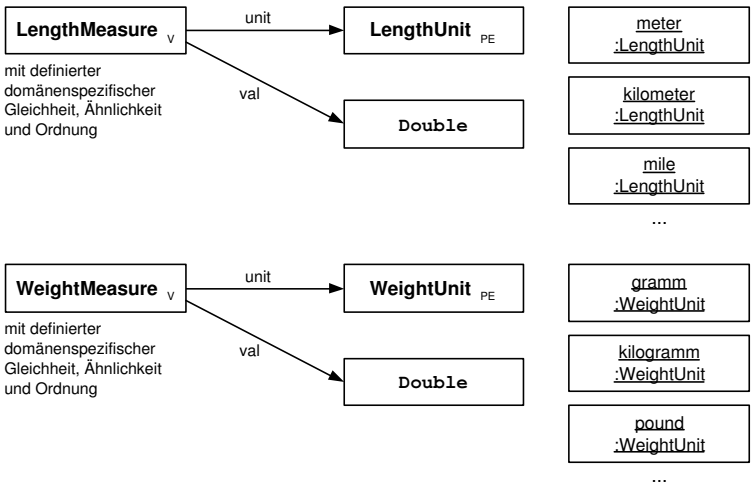
A.7 category.capabilityservice

ONTOLOGY: category.capabilityservice



A.8 domain.measure

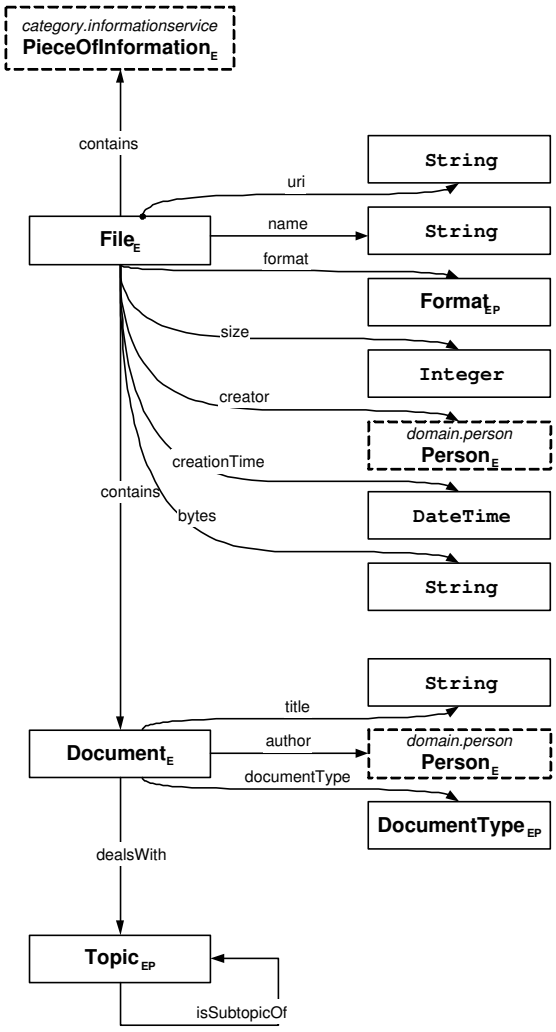
ONTOLOGY: domain.measure



...

A.9 domain.file

ONTOLOGY:domain.file



Literatur

[1] Baader, F., D. Calvanese, D. McGuinness, D. Nardi und P. Patel-Schneider: *Description Logic Handbook – Theory, Implementation and Applications*. Cambridge University Press, 2002.

[2] Fischer, T.: *Entwicklung eines Kontainermanagers zur Unterstützung der Konfigurierung und Ausführung semantisch beschriebener Dienste*, März 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[3] Gruber, T.: *A Translation Approach to Portable Ontology Specifications*. Knowledge Acquisition, 5:199–220, 1993.

[4] Herzog, T.: *Aktive VISIO-Schablonen zur grafischen Erstellung von DIANE-Dienstbeschreibungen*, Oktober 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[5] Höllrigl, T., C. Schaa und F. Schell: *Entwicklung einer formalen Repräsentation für die DIANE-Dienstbeschreibung*, Mai 2004. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[6] Kifer, M., G. Lausen und J. Wu: *Logical Foundations of Object-Oriented and Frame-Based Languages*. Journal of the ACM, 42(4):741–843, July 1995.

[7] Klein, M. und B. König-Ries: *A Process and a Tool for Creating Service Descriptions Based on DAML-S*. In: *Proc. of the 4th VLDB Workshop on Technologies for E-Services (TES'03)*, S. 143–154, Berlin, Germany, September 2003.

[8] Klein, M. und B. König-Ries: *Combining Query and Preference - An Approach to Fully Automatize Dynamic Service Binding*. In: *Short Paper at IEEE International Conference on Web Services*, San Diego, CA, USA, July 2004.

[9] Klein, M. und B. König-Ries: *Coupled Signature and Specification Matching for Automatic Service Binding*. In: *Proc. of the European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, September 2004.

[10] Klein, M. und B. König-Ries: *Integrating Preferences into Service Requests to Automate Service Usage*. In: *First AKT Workshop on Semantic Web Services*, Milton Keynes, UK, Dezember 2004.

[11] Klein, M., B. König-Ries und P. Obreiter: *Stepwise Refinable Service Descriptions: Adapting DAML-S to Staged Service Trading*. In: *Proc. of the First Intl. Conference on Service Oriented Computing*, S. 178–193, Trento, Italy, December 2003.

[12] König-Ries, B. und M. Klein: *Information Services to Support E-Learning in Ad-Hoc Networks*. In: *First International Workshop on Wireless Information Systems (WIS2002)*, S. 13–24, Ciudad Real, Spain, April 2002.

[13] Martin, D. L., A. J. Cheyer und D. B. Moran: *The Open Agent Architecture: A Framework for Building Distributed Software Systems*. Applied Artificial Intelligence, 13(1-2):91–128, January-March 1999.

[14] Müssig, M.: *Schaffung des Vergleichers für DIANE Service Descriptions*, Februar 2005. Diplomarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[15] OASIS, UN/CEFACT: *ebXML*. http://www.ebxml.org/.

[16] Stern, M.: *Generierung von Anwendungsbeispielen für den Vergleicher von DIANE Service Descriptions*, Januar 2005. Studienarbeit an der Fakultät für Informatik, Universität Karlsruhe.

[17] Web-Ontology Working Group: *Web Ontology Language - Services (OWL-S)*. http://www.daml.org/services/daml-s/0.9/.

[18] World Wide Web Consortium: *Web Service Description Language (WSDL)*. http://www.w3.org/TR/wsdl.

[19] World Wide Web Consortium: *XML Schema Part 2: Datatypes*. http://www.w3.org/TR/xmlschema-2/.

Index